UNCLASSIFIED

(2)

IDA PAPER P-2120

A FORMAL SPECIFICATION AND VERIFICATION METHOD FOR THE PREVENTION OF DENIAL OF SERVICE IN Ada SERVICES

Che-Fn Yu
Virgil D. Gligor

March 1988

Prepared for
Ada Joint Program Office (AJPO)

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

UNCLASSIFIED

DEFINITIONS

IDA publishes the following documents to rep results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Group Reports

Group Reports record the findings and results of IDA established working groups and panels composed of senior individuals addressing major issues which otherwise would be the subject of an IDA Report. IDA Group Reports are reviewed by the senior individuals responsible for the project and others as selected by IDA to ensure their high quality and relevance to the problems studied, and are released by the President of IDA.

Papers

Papers, also authoritative and carefully considered products of IDA, address studies that are narrower in scope than those covered in Reports. IDA Papers are reviewed to ensure that they meet the high standards expected of refereed papers in professional journals or formal Agency reports.

Documents

IDA Documents are used for the convenience of the sponsors or the analysts (a) to record substantive work done in quick reaction studies, (b) to record the proceedings of conferences and meetings, (c) to make available preliminary and tentative results of analyses, (d) to record data developed in the course of an investigation, or (e) to forward information that is essentially unanalyzed and unevaluated. The review of IDA Documents is suited to their content and intended use.

The work reported in this document was conducted under contract MDA 903 89 C 0003 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

This Paper has been reviewed by IDA to assure that it meets the high standards of thoroughness, objectivity, and appropriate analytical methodology and that the results, conclusions and recommendations are properly supported by the material presented.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1988	Final

4. TITLE AND SUBTITLE
A Formal Specification and Verification Method for the Prevention of Denial of Service in Ada Services

5. FUNDING NUMBERS
MDA 903 89 C 0003

T-D5-304

6. AUTHOR(S)
Che-Fn Yu. Virgil D. Gligor

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Defense Analyses (IDA)
1801 N. Beauregard Street
Alexandria, VA 22311-1772

8. PERFORMING ORGANIZATION REPORT NUMBER
IDA Paper P-2120

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office (AJPO)
Room 3E114, The Pentagon
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, unlimited distribution.

12b. DISTRIBUTION CODE
2A

13. ABSTRACT (Maximum 200 words)
The Institute for Defense Analyses was asked by the Ada Joint Program Office and the Rome Air Development Center to review the denial-of-service problem and introduce a new formal specification and verification method for the prevention of denial of service. A formal method for establishing the specification-to-code correspondence was used. This enabled the authors to verify formally the prevention of denial-of-service in Ada services. To verify the absence of denial of service, a service specification model is introduced. A key component of that model is the separation of the service sharing mechanism from the service sharing policy. The argument is that, in contrast with other properties, the prevention of denial-of-service requires specification of service use, i.e., user agreements which external constraints on service invocations and which must be obeyed by all service users.

14. SUBJECT TERMS
Ada Programming Language; Formal Specification; Verification; Denial of Service; Service-Sharing Mechanism; Computer Security.

15. NUMBER OF PAGES
94

16. PRICE CODE

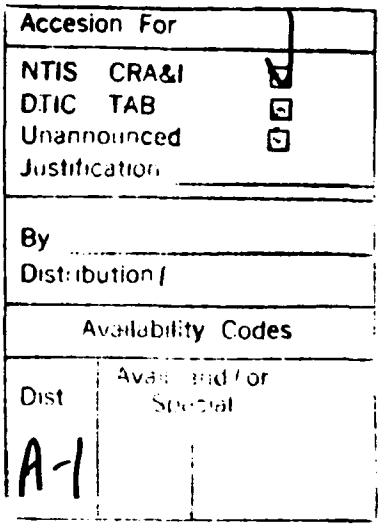
17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

IDA PAPER P-2120

A FORMAL SPECIFICATION AND VERIFICATION METHOD FOR THE PREVENTION OF DENIAL OF SERVICE IN Ada SERVICES

Che-Fn Yu
Virgil D. Gligor

March 1988



INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 89 C 0003
Task T-D5-304

Contents

List of Figures

Preface

In this paper we review the denial-of-service problem and introduce a new formal specification and verification method for the prevention of denial of service. We also use a formal method for establishing the specification-to-code correspondence. This enables us to verify formally the prevention of denial of service in Ada services written using the package and tasking semantics. We also illustrate the use of the formal specification and verification proofs in the formal verification and proofs of denial-of-service prevention for Ada services.

To verify the absence of denial of service, a service specification model is introduced. A key component of that model is the separation of the service sharing mechanism from the service sharing policy. The need for specifying fairness and simultaneity conditions formally within the sharing policy is discussed. We argue that, in contrast with other properties, the prevention of denial of service requires specifications of service use; i.e., user agreements, which are external constraints on service invocations and which must be obeyed by all service users. In general, these constraints cannot be converted into internal service-enforced constraints, such as those for service-sharing mechanisms and policies. We show that the formal specification of sharing policy and that of user agreements form the basis for proof of denial-of-service prevention. We also explain why previous methods developed for verification of liveness and safety properties of concurrent programs cannot be used directly to demonstrate absence of denial of service in shared services. We illustrate the difference between denial of service, security, and integrity problems and point out that formal specification and verification methods developed for these latter two areas cannot be used to demonstrate absence of denial of service.

This paper assumes that the reader is a computer science or engineering professional working in the area of formal specification and verification of security and concurrency properties of code. Numerous literature citations are made to temporal logic, and familiarity with temporal logic and with its prior use to solving safety and liveness problems of concurrent programs is assumed. Also, familiarity with Ada semantics, primarily with the notion of the package and of tasking, is necessary to understand the formal verification of service written in Ada.

Acknowledgements

The authors would like to thank Dr. Brent Hailpern, Mr. William T. Mayfield, and Dr. Robert I. Winner for their contributions. The sharing of their technical expertise through comments and suggestions has enhanced our work.

1. INTRODUCTION

The three major security concerns, namely unauthorized release of information, unauthorized modification of information, and denial of service, have been addressed to various degrees during the last decade. For example, security policies and models of authorized release of information exist and seem to proliferate at a rapid pace. Experience with the formal/informal interpretation of these models in real systems has also been accumulated to the extent necessary to determine both the models' and their interpretations' strengths [Landwehr84, Millen84]. Relatively few models of authorized modification of information exist, and a consensus on their usefulness seems to be more difficult to reach.

In contrast with the areas of (un)authorized release and modification of information, the denial-of-service area has received little attention in the past. Formal models are conspicuous through their absence and, until recently not even an acceptable general definition of the problem had been proposed [Gligor83]. The properties of this problem are not well known and misconceptions about it abound among the security practitioners. In fact, few of these practitioners even consider denial of service a security problem.

The principal goal of this paper is the presentation of a new formal specification and verification method for the prevention of denial of service and its application to shared services written in the language Ada. These services use the semantics of the "package" and "tasking" mechanisms of Ada. An additional important goal is the illustration of the practical use of this method to the formal verification of denial-of-service prevention in Ada services.

The formal specification and verification method presented here is the only method developed to date which demonstrates the prevention of denial of service. In this paper we also explain why the previous methods developed for the verification of liveness and safety properties of concurrent programs cannot be used directly to demonstrate absence of denial of service in shared services. Also, by explaining the difference between denial of service, security, and integrity problems we point out that formal specification and verification methods developed for these latter two areas cannot be used to demonstrate absence of denial of service.

This paper contains seven sections and two appendices. In the reminder of the introduction we review the denial-of-service problem and its definition. In Section 2 we explain the differences between denial of service, security, and integrity problems. In Section 3 we explain the relationship between the denial-of-service properties and the safety and liveness properties in concurrent programming. We also argue that current formal methods that help demonstrate the absence of safety and liveness problems in concurrent programs are insufficient to demonstrate absence of denial of service in shared services that can be invoked concurrently. In Section 4 we introduce a new formal specification and verification method that helps prove the absence of denial of service in shared services. The underlying concepts of this method, such as the model of shared service and the notion of user agreements, are illustrated through a detailed example.

Service and user-agreement specification are provided using a language based on temporal logic. A complete formal specifications of a generic resource allocator service that can be invoked concurrently is introduced and the proof of denial-of-service prevention is provided. In Section 5 we illustrate the use of the formal specification and verification method in the development of formal proofs of denial-of-service prevention in services written in Ada. The specification-to-Ada code mapping is presented for the resource allocator example, and the mapping of the formal specification proofs into the formal correctness proofs on Ada code are also discussed. Section 6 of this paper contains the conclusion and Section 7 contains the references. The Appendices include a summary of temporal logic semantics and a list of derived temporal theorems.

1.1. REVIEW OF THE DENIAL-OF-SERVICE PROBLEM AND DEFINITION

In order to define the denial-of-service problem precisely we need to introduce the problem's entities involved and the relationships among them. The *shared service* is a generic term that includes programmed access to shared data, invocation and execution of programs, and use of hardware resources. The designer of each shared service usually specifies and implements the service sharing mechanism and policy. This mechanism enforces conventions of service use such as the service-invocation procedure, entry point, number, type, and order of parameters. The policy determines the maximum number of users, the order of service use in cases of competitive demand by users, and specifies the intended waiting time for the service in case of competitive demand. In addition to the mechanism and policy specification, the service specification may also include a description of the service results as a function of its input parameters.

The waiting time for a service is rarely specified explicitly in terminal-oriented, time-sharing operating systems; i.e., through stated specifications such as "any requesting user will wait no more than 'x' units of time before the service is granted." Such specifications are typical for real-time system services. Most often, the waiting time is specified implicitly, through unstated specifications which, nevertheless, imply that "any requesting user will *eventually* be granted the service;" i.e., the user will be granted service in a finite time. This type of waiting time specification is the one assumed throughout this paper. The conditions that specify the finite waiting time for a service also define the Finite Waiting Time (FWT) policy. We also note that:

- if the waiting time $\rightarrow 0$, there can be no denial of service because, in essence, the shared service becomes private, and
- if the waiting time $\rightarrow \infty$, there can be no denial of service because, by definition, the shared service is not promised to any user.

For the purposes of this paper, the users are processes which execute instructions on behalf of the human user. Operating system processes may also be users of various internal operating system services. The users rely on the service specifications and want to ensure that, among other things, the service is available within finite time. Users may also care about other service features related to the service-scheduling policy, such as

average service time, service cost, etc., which are not directly related to the denial-of-service problem.

The service itself may consist of one or more processes. These processes, in turn, may be users of other services. The only constraint we impose on the relationships among services is that the "use" relationship [Parnas76, Parnas79] should not lead to cycles – a situation which can always be detected by careful analysis of the operating system [Neumann et al.75, Schroeder et al.77, Janson76, Haberman76, Osterhout80].

Definition 1. (Denial of Service) A group of authorized users of a specified service is said to deny service to another group of authorized users if the former group makes the specified service unavailable to the latter group for a period of time which exceeds the intended (and advertised) service waiting time.

In principle, denial of service can occur only when a group of the service users becomes mores "privileged" than the rest of the users. More privileged users are those who can gain higher priority or a more powerful access to the service. Whereupon the more privileged users can exercise their privileges individually, or in collusion, to prevent other users from accessing a specified service for periods of time exceeding the intended waiting time or any FWT of that service. Such privileges can be acquired by exploiting various discriminatory service-sharing policies and flawed service-sharing mechanisms. The less privileged users become dependent upon the behavior of the more privileged users.

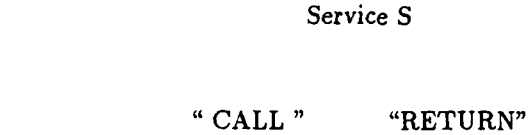
Definition 2. (Interuser Dependency) A user of a shared service is dependent upon another user of that service if the correct (i.e., intended and advertised) operation of the former requires the correct operation of the latter.

Some interuser dependencies are legitimate and have desirable effects; e.g., the sharing of some object provided by service.

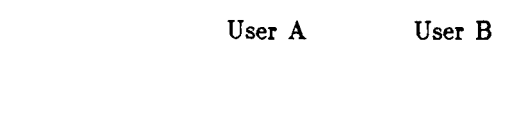
Definition 3. (Legitimate Dependency) An interuser dependency is legitimate whenever it results from user-visible object sharing within the service (and which is explicitly identified within the service specification).

For example, if two users share a file within a file-access service, the two users are dependent upon each other because the data written by one user may affect the correct operation of the other and vice versa (Figure 1). Furthermore, the concurrency control mechanism used by the service to serialize access to the shared file may, in some cases, introduce interuser dependencies and cause denial of service. Deadlocks as well as user attempts to abort their processes (actions which may trigger recovery) may result in delays that can cause denial of service.

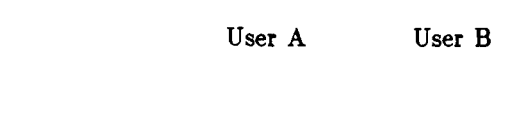
Similarly-defined interservice dependencies are legitimate because they form the basis for hierarchical service development. For example, the "call" instruction introduces interservice dependencies because the caller service always relies upon the called service to provide correct results [Parnas76, Parnas79]. This is illustrated in Figure 1. Other interservice relations–not just the one defined by the "call" instruction–have been identified and analysed in references [Neumann et al.75, Schroeder77, Janson76, Haberman76]. Failures of a service to meet its specifications cause failures in all services which depend upon it. For example,



(a) Interservice Dependency – Legitimate (S α T).



(b) Interuser Mutual Dependency – Legitimate (A α B, B α A).



(c) Interuser Undesirable Dependency (B α A).

Fig. 1. Dependency Examples (A α B $\equiv$ A depends on B).

hardware failures due to component decay can cause operating system services to stop functioning causing a system crash. The crash recovery action results in delays which can cause denial of service.

Legitimate denial of service is a temporary phenomenon which can be dealt with in two ways. The delays in recovery from such events as deadlocks and crashes can be accounted for in the computation of the waiting time for each service in the service hierarchy. Although this computation can be nontrivial, it is always possible. Thus, legitimate denial of service can be ***prevented by definition.*** This approach is advisable whenever the recovery delays are, or can be made, relatively small, i.e., less than a few seconds. For example, recovery from crashes caused by hardware failures can be made very fast by adding redundant hardware components, failure detection and reconfiguration logic.

In some cases, however, legitimate denial of service cannot be prevented by definition. The cost of masking a failure and making recovery fast, or that of deadlock prevention or avoidance may be too high. In cases when legitimate denial of service cannot be prevented by definition, denial of service may be allowed to occur. The system and/or the users are relied upon to detect and signal such occurrences.

Detection of denial of service is always possible provided that a Maximum Waiting Time (MWT) is specified. Service invocations by users can be designed to trigger, in an atomic way [Lampson81], the setting of a timer to the sum of the MWT and the maximum service time. Various methods have been routinely used by designers of real-time systems to compute the maximum service time and, subsequently the MWT [Gouda79]. Whenever denial of service is detected, the system may be asked to inform the users of the extent of the (temporary) denial of service, and proceed to recover from it.

Recovery from denial of service may require that a user invokes an alternate service with similar specifications to those of the service being denied. The alternative of notifying the user of denial-of-service instances is advisable in case of longer legitimate delays; it also seems to be the one preferred by most operating systems designers.

Definition 4. (Undesirable Dependency) Any dependency introduced between unrelated users (i.e., users that do not share explicitly any objects within a service) by service mechanisms and/or policies is ***undesirable.***

For example, a user process which manages to exhaust a resource bound within a service makes other users dependent upon itself. The dependency occurs because no other user can access the service until the first user relinquishes at least a part of the resource. Consequently, no user–except possibly the first one–may operate correctly; e.g., neither the users nor the service can meet their intended specifications, such as their maximum execution time.

Some undesirable interuser dependencies may appear along with some of the legitimate dependencies within a service. Therefore, all solutions to the denial-of-service problem (1) must distinguish between legitimate and undesirable interuser dependencies within services, (2) must provide detection/recovery mechanisms for denial-of-service instances that might be caused by legitimate dependencies, and (3) must eliminate undesirable dependencies.

In reference [Gligor83] it is shown that the sources of undesirable dependencies can be traced to discriminatory service-sharing policies, inadequate service-sharing mechanisms, and to combinations of seemingly-adequate service-sharing mechanisms and policies. For example, the "shortest-job-time-first" policy for processor scheduling and the "shortest-seek-time-first" policy for disk scheduling are discriminatory sharing policies.

Among the inadequate service-sharing mechanisms the following have led to the largest numbers of undesirable interuser dependencies (and consequently to denial-of-service instances): inadequate resource quotas, inadequate access control mechanisms, inadequate concurrency control mechanisms, and inadequate combinations of mechanisms. Lack of resource quotas, circumvention of resource quotas, and inadequate handling of resource-quota exceptions are the most common sources of undesirable dependencies [Gligor83]. Weak access control mechanisms (i.e., those which do not violate controlled sharing policies) may allow built-in system dependencies on users' behaviour within services. These dependencies can also cause undesirable interuser dependencies. Interuser timing dependencies, such as those leading to individual process blocking, are the most common sources of undesirable dependencies caused by inadequate concurrency control [Gligor83].

2. DENIAL OF SERVICE VS SECURITY AND INTEGRITY PROBLEMS

In the past few years several misconceptions about denial of service have appeared due to the complexity of the problem and to its many guises. The relationship between denial of service and other security problems and the relationship between denial of service and service availability/reliability problems must be clarified before significant progress can be made towards resolving these misconceptions. Also, the nature of a Trusted Computing Base that prevents denial-of-service instances must be explained. In this section, several misconceptions about these issues are addressed in the form of answers to relevant questions.

(1) Is denial of service a security problem?

The answer to this question is unequivocally yes, for at least two reasons. Denial of service can result in unauthorized disclosure of information. This is the case because ostensibly-confined user programs can leak information to other user programs by preventing the latter's access to a legitimately-shared service in an observable way.

This observation is more important than it first appears. A large number of denial-of-service instances are due to lack of resource quotas for various resources. It would be ironic to solve the denial-of-service problem by placing resource quotas in an otherwise secure system, only to discover that a large number of covert channels have been introduced. It should not be surprising that the covert channel problem and the denial-of-service problem are related; this is a consequence of the fact that both these problems appear only in the context of shared resources.

The second observation is that the opposite of the denial-of-service problem, namely the policy of guaranteed access to data, programs and other services by authorized users, is a problem of controlled-sharing of information. Because security must handle controlled-sharing problems, denial of service must be also considered a security problem.

(2) Is denial of service a distinct security problem, or is it a special case of the disclosure or of the integrity (i.e., unauthorized modification of information) problem?

Denial of service is a distinct security problem, therefore, it should not be expected that it will disappear by inventing new disclosure or integrity policy models. This statement is based on the following line of reasoning. Unauthorized disclosure of information does not constitute denial of service. Conversely, it could be argued that any denial-of-service instance constitutes a disclosure problem because all such instances can provide at least a single, unauthorized, binary answer to an arbitrarily complex question. This can happen because a user, which is prevented from answering a query from some other users by the security policies and mechanisms of a system, may nevertheless provide a binary answer to the query by denying (or not denying) access to the other user to a shared service at predetermined times. Thus, one may argue that denial-of-service problem is a subset of the unauthorized disclosure problem and, therefore, denial of service is not a distinct security problem. However, this argument is misleading for at least two reasons. First, it is

also known that not all denial-of-service instances cause integrity problems; but all integrity problems cause some form of denial of service because some information is corrupted and, therefore, denied to some users [Gligor83]. Thus, it would follow that all integrity problems are a subset of the denial-of-service problem, and transitively, that all integrity problems are a subset of unauthorized disclosure problems. Therefore, since we consider integrity a distinct problem from the disclosure problem, the denial-of-service problem must also be considered to be a distinct problem. Second, from a more practical point of view, none of the current models that define disclosure policies can also define denial-of-service/guaranteed access policies, and none of the current integrity models can handle all aspects of denial of service [Millen 84]. Thus, we can conclude that the denial-of-service problem is a separate security concern that appears even when all users are legitimately *authorized* to access and modify data.

(3) Should a practical definition of denial of service be based on the notion of a system crash?

The temptation of defining denial of service in terms of system crashes should be resisted. First, a crash may or may not cause denial of service. For example, in a network of computer systems, the crash of one system does not necessarily imply the crash of the entire system. Thus, a user may be able to initiate and complete his computation within the necessary maximum waiting time advertised by the system. The crash becomes merely a failure of system component, and the potential denial of service can be prevented by invoking alternate services, as discussed in Section 1.1 above.

Second, denial of service does not necessarily cause a system crash. Examples of denial-of-service instances which do not cause crashes either in centralized systems or in networks are presented in [Gligor83]. In fact, it is argued in [Gligor84] that denial of service is not necessarily visible to observers outside of the group being denied service. This implies that reliance solely on system operators to detect and recover from instances of denial of service is inadequate. (This approach is attributed by some [Wilkinson82] to Burroughs large systems and by [Ritchie78] to UNIX.)

Third, the notion of a crash is itself an imprecisely defined notion in most systems. Few systems distinguish precisely between recoverable failures and disasters (i.e., failures that always lead to crashes). While such a distinction is an important concern of reliable system design, it is not a direct concern of secure system design. However, the recovery of a system in a secure state is an important security concern which is, quite obviously, unrelated to denial of service.

(4) Is denial of service a fundamental problem or is it just an isolated design/ implementation problem? How important is the denial-of-service problem?

If one defines as "fundamental" a problem that persists when the cost of technology decreases to zero, then the denial of service is a fundamental problem. Denial of service appears because users share local or remote services, and controlled service-sharing remains an important concern even when the cost of technology decreases to zero. First, services are not only shared because of the high cost of using them privately; sometimes they are shared because the service supplier is unique (e.g., no other supplier is creative

enough to provide the required service). Second, the need to share information among several users may require the sharing of services that make much information available. Thus, denial of service is not an isolated design/implementation problem that appears in only some systems. It appears in all designs and implementations where services are shared.

Denial of service, although a fundamental problem, is only of secondary importance currently in both the security and the high-availability areas of research and development. This is the case because, as suggested above, a large number of denial-of-service (but not all) instances are eliminated (1) by the use of integrity models for secure systems, and (2) by the use of designs for high-availability systems. However, when such models and designs become better understood, the importance of the denial-of-service problem will become clear.

(5) Are the current Trusted Computing Bases (TCB) [DoD83] structured adequately for the prevention/detection of denial of service?

The answer to this question appears to be no. First, it is well known that a TCB which implements a disclosure and integrity model may, in fact, not address the denial-of-service problem. For example, the separation of policy from mechanism in kernel design, which was considered a cornerstone for good kernel design in the UCLA Secure UNIX and in the Hydra systems [Popek75, Levin75], places the various resource-allocation-policy modules outside the kernel and trusted processes. These systems admittedly do not address the denial-of-service problem. However, these examples serve to illustrate the contention that adequate structuring of the TCB for authorized disclosure and data integrity purposes may be still inadequate for addressing the denial-of-service problem.

Two misconceptions about TCB structuring for disclosure and integrity purposes are likely to propagate to the TCB structuring for denial-of-service purposes. First, the notion that security kernels always are (or can be made) small is mistaken. Practice shows that complex systems have large kernels and simple systems have small kernels [Janson76]. Second, the notion that for a given system there exists an unchanging TCB is also mistaken. The size of a TCB, at least in terms of the number of trusted processes, grows with the number of applications. This is a particularly important observation in the case of denial of service because shared, application-level services that are implemented outside a disclosure/integrity TCB may, in fact, have substantial parts of their implementation inside a denial-of-service-oriented TCB.

3. DENIAL OF SERVICE VS SAFETY AND LIVENESS PROBLEMS

Various *safety* and *liveness* properties of shared services or program modules are widely used in the formal specification and verification of concurrent programs [Owicki82], [Hailpern82], [Lamport83], that denial-of-service problems have been eliminated from shared services. It should be noted that flawed specifications of concurrency control mechanisms and/or policies are not the only source of denial of service [Gligor83]. Also, safety and liveness problems that are unrelated to service sharing are irrelevant to denial of service. In this section, we explain the relationship between denial of service and safety/liveness problems.

3.1. DENIAL OF SERVICE AS A SAFETY/LIVENESS PROBLEM

Denial of service can be both a safety and a liveness problem. It takes place whenever one or both of the following situations occur:

- Some users prevent some other users from making progress within the service for an arbitrarily long time.
- Some users make some other users receive incorrect service; i.e., the service does not satisfy its intended functional specifications for the latter users. The service is disabled in an unauthorized way.

Denial-of-service instances of the first case are liveness problems, whereas denial-of-service instances of the second case are safety problems. Note that permanent deadlocks between user processes is considered to be a safety problem.

A user is said to make progress within a service if all of its service invocations will eventually terminate. Intuitively, a service allows its users to make progress whenever its sharing policies are fair, whenever it is free from permanent deadlock, and if it is free from *starvation*. Fairness and freedom from *starvation* are liveness concerns. Furthermore, a service performs correctly whenever different user operations within the service neither interfere with each other nor disable the service, and whenever the effects of a user operation within the service always persist after the operation is committed by its user. All these properties are safety concerns.

3.2. DENIAL OF SERVICE AS A DISTINCT SAFETY/LIVENESS PROBLEM

The major *safety* properties that have been specified and verified formally in the concurrent programming area are mutual exclusion, resource invariance, and deadlock freedom. The major *liveness* properties that have been formally specified and verified in a general way are termination and fairness. *Starvation freedom* is also a liveness property that has been verified formally in some examples. These *safety* and *liveness* properties are the fundamental properties of concurrency control mechanisms within shared services. Since services may be invoked concurrently, these properties must be used to prevent denial-of-service problems.

However, denial-of-service problems can also be caused by some other sharing problems unrelated to safety or liveness. Failure to use adequate service sharing mechanisms, such as resource quotas, user identification, and service policies such as the finite waiting time for individual users, can cause denial of service. For example, inadequate enforcement of resource quotas may stop the activity of the service [Gligor 83-84]; lack of identification checking may cause resource inconsistency [Welsh 81]; lack of finite-waiting-time specification may deny service to some users even if a fair scheduling policy is enforced within the service [Yu 87].

One of the reasons why denial of service differs from the typical safety/liveness problems is that it may result from inadequate use of services by users. Undesirable user behavior can cause some users to receive incorrect service or prevent other users from making progress within the service. These types of denial of service may occur because shared services have no control over the users' behavior outside the service. For example, the service behavior may depend on the order of the entry operations invoked by users. Although service properties related to the order of service invocations are, in general, safety properties of a given service, violations of such properties may be caused by violation of liveness properties in services using the given service. Whenever a service assumes a certain invocation order and the assumed order is not guaranteed, then denial of service may take place. For example, a malicious user can monopolize a resource-providing service by requesting a large number of resources without releasing them for an arbitrarily long time. Although some aspects of user behavior can be checked by other system facilities outside the service (e.g., through compilation checks), a resource can become unavailable for an arbitrarily long time when the user holding the resource gets blocked within another service; or when the user aborts before normal service termination and some of the service operations cannot finish their work on behalf of other users.

Notice that current service models, such as those based on monitors [Hoare 74] and resource controllers [Ramamritham 85], which are usually used for synchronizing asynchronous users, are unable to protect against the denial-of-service problems caused by undesirable user behavior. This is because, although current service models can always schedule the proper order of operations among different users, they cannot always control the invocation order of individual users. Current service-specification models focus on the integration of concurrency control into program modules of services, and thus are irrelevant to the user behavior outside the service.

4. FORMAL SPECIFICATION AND VERIFICATION METHOD

In this section, we introduce a formal specification and verification method that is suitable for the prevention of denial of service in computer systems. First, we address the assumptions and properties of shared services under consideration. Second, we introduce the notion of "user agreement". Third, we present the formal model for service and agreement specification. Fourth, we apply the formal model to specify the resource allocator and verify the prevention of denial of service in the resource allocator. Finally, we justify our formal method and discuss why other methods are insufficient for the prevention of denial of service.

4.1. ASSUMPTIONS

The relations among shared services are specified in the following two axioms:

1) Hierarchical structure axiom: the call relations among shared services are partially ordered; i.e. no cycles on all "call" sequences may exist in the systems under consideration.
2) Independence axiom: the sharing mechanism and policy of all services in a system are independent of each other; i.e., they are uniquely determined in each service.

As mentioned before, denial of service occurs if and only if *undesirable interuser dependencies* exist; thus the elimination of *undesirable interuser dependencies* is sufficient to prevent denial-of-service problems. However, the elimination of *undesirable interuser dependencies* is not trivial for complex systems especially whenever the system shared services are not well organized. The above assumptions relating to the shared service enable us to identify the *undesirable interuser dependencies* among users of each individual service without analyzing other services that may be called from within the services under consideration. The hierarchical structure axiom eliminates the mutual dependency between operations in a "call" sequence. Without this axiom, it would appear to be very difficult to analyze the undesirable interuser dependencies. The independence axiom makes the undesirable interuser dependencies which may exist within a specific service, independent of those of other services which may be called from within the specific service, if all of these calls are terminated successfully. These assumptions are also general requirements for constructing well-structured systems such as the *monitor* or the *abstract type* based systems.

4.2. PROPERTIES OF SHARED SERVICES

The shared services under consideration are defined by several sharing-related properties as follows:

- The extent of shared services: Services can be shared locally, remotely, or distributively. Local and remote services are shared to minimize the high cost of using resources privately. Typically, distributed services are shared in to increase the availability of resources. All shared services can be accessed concurrently; thus the specification and use of concurrency control mechanisms and fair sharing policies is required for all types of shared services.

- Separation of sharing mechanisms and policies from other functionally related mechanisms: Sharing mechanisms and policies are used for controlled sharing of the service-provided resources, whereas other functional mechanisms are used for actual manipulation of the resources. Sharing mechanisms and policies regulate the execution of functional operations and thus are the main source of undesirable interuser dependencies which cause denial of service. In contrast, functional mechanisms determine how to use the resource and this is unrelated to sharing control; thus they are not the source of undesirable interuser dependencies within the service. Thus, our service model will emphasize the sharing control part of the shared services in order to facilitate our analysis of denial-of-service problems, whether concurrency is actually used or not. Other service models, such as those for monitors [Hoare 74], and for concurrent programs [Hailpern 82], do not attempt to separate the sharing-control part from the functional part of shared services. This is because the problems caused by sharing mechanisms and policies other than concurrency control are irrelevant to those models.

- Non-uniform sharing mechanisms and policies: Depending on the service environment, sharing mechanisms and policies are implemented in different ways in different services. Consider synchronization concerns within a service. The appropriate implementation of synchronization mechanisms can differ not only for different services, but also for similar services in different environments. For example, traditional operating system synchronization mechanisms are implemented based on *operation atomicity* in the sense that an invocation to a service is considered as a complete unit of work by itself. Thus, simple mechanisms, such as the mutual exclusion mechanism, are sufficient. In contrast, synchronization mechanisms for distributed services require *transaction atomicity* to ensure failure atomicity during the evolution of a transaction [Gray 78]. During execution, a transaction evolves as a partial order of a set of related operations on objects of a distributed service to ensure data integrity of the service. Thus, more complex mechanisms, such as the *two-phase locking* mechanism, are required.

- System level vs. application level services: The sharing mechanism and policies for system level service are different from those of application level service. For example, consider the resource allocation services. Deadlocks are often allowed in application level services because the use of resources may not be determined with certainty. In contrast, *deadlock* must be prevented in system level services. Thus,

deadlock detection and recovery mechanisms are required in application level services, whereas certain constraints on user's behavior to ensure the proper order of invocation to entry operations are required in the use system level services to prevent the occurrence of *deadlocks*.

4.3. THE NOTION OF USER AGREEMENT

Denial of service can take place not only by exploiting flawed sharing mechanisms and/or policies within the service but also by undesirable sequences of service invocation outside the service. Since these constraints include invocation sequences of multiple service users, they are called the "user agreements". User agreements for service access must be specified outside the service specifications for two reasons. First, user agreements must eliminate the undesirable interuser dependencies created by user misbehavior outside the service. (Other undesirable interuser dependencies created by inadequate sharing mechanisms and policies can also exist [Gligor83].) Second, user agreements may constrain the use of multiple services, and therefore cannot be included into any single service specification. For example, Havender's "ordered resource acquisition" approach to deadlock prevention [Havender68] is an example of a user agreement that spans multiple resource-providing services inside an operating system.

The specification of user agreements outside a service does not necessarily mean that service users are trusted to obey them. Whenever compile-time checks on user code are impractical, the user agreements may be enforced by code outside the service that is executed before the service calls are actually issued. An example of such code in other areas, such as in the deadlock avoidance area, is that which determines whether a resource assignment to a user is safe [Habermann 69]. Another example is the code which enforces that lock requests issued by users are legal, well-formed, and two phase [Gray78].

To define the user agreements for shared services, we have to analyze all possible invocation sequences that may be issued by each user and all possible invocation sequences that may invoke the shared service. For convenience, the former sequence is called the *U_seq* and the latter is called the *S_seq*. A U_i*_seq* is a partial order of service invocations issued by an individual user U_i. A *S_seq* is a partial order of concurrent service invocations by many users. Thus, a *S_seq* is an invocation sequence that interleaves operations of individual *U_seqs* and preserves the original partial order of each *U_seq*. Analysis of *U_seq* and *S_seq* allows us to define user agreements for shared services.

4.3.1. Safe Service-Invocation Sequence

Operations needed for controlled service sharing can be classified into two categories: (1) the resource consuming operations and (2) the resource producing operations. For example, the *Acquire* and *Release* operations of resource allocators can be considered the consuming and producing operations, respectively. Similarly, the *Put* and *Get* operations of a bounded buffer service are the consuming and producing operations, respectively. Some services, such as the resource allocators, require their users to produce resources that have been previously consumed by the same users. Other services, such as the bounded-buffer services, allow resources to be consumed and produced by different users. Therefore, we have to define the allowed types of operations and the allowed order of operations for each user. Let a_i be the allowed consuming operation invoked by user U_i, and b_i be the allowed operation that attempts to produce the resources consumed by user U_i (note that b_i is not necessarily invoked by U_i). The set of allowed operations for U_i can be defined as:

$$Op_i\_set = \{op \mid op \in A_i \text{ or } op \in B_i\}$$

where A_i represents the set of all allowed a_i, and B_i represents the set of all allowed $b_{j1}, b_{j2}, \cdots$ that produce resources consumed by users $U_{j1}, U_{j2}, \cdots$ respectively. For resource allocators, the set of allowed operations is $Op_i\_set = \{Acquire_i, Release_i\}$, and each user U_i is allowed to invoke both operations. For bounded-buffer services, there are two types of users; viz., senders and receivers. The set of allowed operations for the sender U_i is $Op_i\_set = \{Put_i\}$, and for the receiver U_j is $Op_j\_set = \{Get_i\}$, where U_i is the peer sender of the receiver U_j.

The allowed invocation order between two operations can be defined by the partial order relation "$\leq$". The order relation specifies that an invocation of operation op_1 must precede an invocation of operation op_2 and is represented as $op_1 < op_2$. Given the set of all allowed operations of user U_i (i.e., Op_i_set), the set of all allowed invocation orders between every two operations in Op_i_set is represented as Or_i_set. In the example of the resource allocator (presented in Section 4.5 below), resources that can be relinquished by user U_i must be those which are previously acquired; thus for U_i, $Or_i\_set = \{(Acquire_i < Release_i)\}$. For bounded-buffer services, no ordering constraints exist for its users because the senders may only invoke *Put* operations whereas the receivers may only invoke *Get* operations. Thus for each user U_i, $Or_i\_set = \{\ \}$ (an empty set). To define a safe user invocation sequence, we use the following notation:

Given $(op_1 < op_2) \in Or_i\_set$

$\widehat{U}_i(k)$ is partial sequence of U_i_seq up to, and including, $U_i(k)$;

$n_{op1}(k)$ is the number of operations op_1 in $\widehat{U}_i(k)$,

$n_{op2}(k)$ is the number of operations op_2 in $\widehat{U}_i(k)$.

Given Op_i_set and Or_i_set, the safe user invocation sequence for U_i can be defined as follows:

Definition:

A user invocation sequence U_i_seq of a specific service is said to be safe if it satisfies the following conditions:

1) *if operation op is in U_i_seq, then $op \in Op_i\_set$, and*

2) *if $(op_1 < op_2) \in Or_i\_set$, then $n_{op1}(k) \geq n_{op2}(k)$ for all k.*

For example, suppose that the producing operation b of the resource allocator presented in Section 4.5 below produces an amount of resources equal to that consumed by the consuming operation a. Then the following U_i_seq is safe:

$$\sigma_i = a_i, b_i, a_i, a_i, b_i.$$

In this U_seq, some resources allocated to user U_i are not relinquished before U_i terminates execution (more a_i's than b_i's). However, user U_i neither relinquishes resources before it acquires them nor does it relinquish resources allocated to other users. Therefore, the user invocation sequence is safe in the sense that it will not cause resource inconsistencies. For the same resource allocator, the following U_i_seqs are not safe:

$$\sigma_i = a_i, b_i, a_i, b_j,$$

$$\sigma_i = a_i, b_i, b_i, a_i.$$

The first U_i_seq is not safe because user U_i attempts to relinquish resources allocated to user U_j. The second U_seq is not safe because user U_i attempts to relinquish its allocated resource twice. Both of these two U_seqs may cause resource inconsistencies.

4.3.2. Live Service Invocation Sequence

Let α be a possible S_seq of a specific service, then the elements of α are the operations that are issued to the service by all users. The order of operations in α is the same as the real time order of all service invocations. If a_i and b_i represent the consuming and producing operations respectively as defined above, then the following sequence is a possible S_seq:

$$\alpha = a_1, a_2, a_3^*, b_1, a_4, a_5^*, b_2, b_5, b_3, \cdots.$$

where a_i^* represents a consuming operation of user U_i which blocked within the service waiting for certain service *conditions* to become true. *Conditions* are boolean functions of service states such as the resource state. *Conditions* may change value only through "calls" to the entry operations. For example, in the invocation sequence given above, the operation a_5, invoked by user U_5, is blocked waiting for some resources to become available. After the resource consumed by user U_2 is relinquished, U_5 resumes its operation (since operation b_2 is immediately followed by operation b_5 and b_5 can appear in the S_seq only after a_5 resumes). For convenience, we use the following notation:

$p_i^*(c)$ is an operation p_i that is blocked for *condition* c,

$p_i(c)$ is an operation p_i that resumes execution after being blocked for *condition* c.

When *condition* c become true, $p_i^*(c)$ might not become $p_i(c)$ immediately due to the resumption of other blocked invocations that are waiting for the same *condition* c. However, a *condition* may become true several times during the evolution of a specific *S_seq*. Given *condition* c for operation p_i, let α_0 be the sub-sequence of a *S_seq* α from the beginning of α to the blocked operation $p_i^*(c)$. Suppose that α_j represents the sub-sequence of α between $(j-1)$-th to j-th time that *condition* c becomes true after $p_i^*(c)$. Then for operation p_i:

$$\alpha = \alpha_0, \alpha_1, \cdots, \alpha_j, \cdots.$$

We use the following notation:

$$p_i^*(c) \xrightarrow{\alpha_j} p_i(c)$$

to represent an invocation p_i that is blocked at the end of α_0 and that resumes operation at the end of α_j.

Definition

A service invocation sequence S_seq is said to be live if, for every blocked invocation $p_i^*(c)$, there exists a set of sub-sequences $\alpha_0, \alpha_1, \cdots, \alpha_j$, and $\alpha = \alpha_0, \alpha_1, \cdots, \alpha_j, \cdots$, such that $p_i^*(c) \xrightarrow{\alpha_j} p_i(c)$.

For example, consider a resource allocator of a single resource. Suppose that each b operation makes *condition* c to be true. *Let a_i = Acquire$_i$ and b_i = Release$_i$, then the sequence*

$$\alpha = a_1, a_2^*(c), b_1, b_2, \cdots, a_{2k-1}, a_{2k}^*(c), b_{2k-1}, b_{2k}, \cdots$$

is a live *S_seq* because $a_{2k}^*(c) \xrightarrow{\alpha_1} a_{2k}(c)$ for all $k \geq 1$, where $a_{2k}(c)$ occurs between b_{2k-1} and b_{2k}. The sequence

$$\alpha = a_1, a_2^*(c), a_3^*(c), b_1, a_4^*(c), b_3, a_5^*(c), b_4, \cdots, a_{k+1}^*(c), b_k, \cdots$$

is not a live *S_seq*. This sequence has $a_k^*(c) \xrightarrow{\alpha_1} a_k(c)$ for all $k \geq 3$. However, for $a_2^*(c)$, no α_j exists in α such that $a_2^*(c) \xrightarrow{\alpha_j} a_2(c)$. Thus, operation a_2 is blocked forever.

4.3.3. User versus Service Invocation Sequences

To obtain appropriate specifications for user agreements of a specific service, the analysis of both *U_seq* and *S_seq* is required. Analysis that is limited to *U_seq* is insufficient because *U_seq* provides only information about what users are allowed to do but not what users must do. Furthermore, analysis that is limited to *S_seq* is also insufficient because liveness of a service invocation sequence is meaningless without knowing that all completed operations in the *S_seq* have received the intended services.

Analysis of *U_seqs* cannot determine the liveness property of the entire service invocation sequence for at least two reasons:

- Resources consumed by a user may not necessarily be produced later by the same user. For example, users that do a *P* operation on a certain semaphore may not do a *V* operation on the same semaphore, and vice versa. Thus, the availability of the resources protected by the semaphore cannot be determined by each individual *U_seq*.
- A user may stop execution in the middle of a *U_seq*, and thus some operations, which other users may invoke, cannot finish their work on behalf of their users. For example, some users' invocations may deadlock each other in several services. The occurrence of such deadlocks cannot be predicted by analyzing *U_seqs* separately. Construction of user agreements based on live *S_seqs* solves the two problems described above.

Analysis of *S_seqs* helps establish liveness properties that cannot be provided by *U_seqs*. However, a live *S_seq* does not guarantee that each individual operation can receive its intended service. An invocation may return abnormally repeatedly (e.g., an exception is signaled before normal return), or may return normally with incorrect results whenever sharing control is incorrectly specified. Therefore, the analysis of both *U_seq* and *S_seq* is required to determine appropriate user agreements of shared services. Of course, the appropriate *U_seq* and *S_seq* depend on the sharing mechanisms and policies within the service under consideration.

4.4. THE MODEL OF SHARED SERVICES

The service-specification model includes two major parts: the service specifications and the user-agreement specifications. The service specifications describe all the desired operations and properties that must be provided by the shared service. The user-agreement specifications describe all the desired properties that must be provided by the users of the shared service.

Given a specific service, the existence of undesirable interuser dependencies (and thus, the potential for denial of service [Gligor83-84]) is determined by three major concerns:

1) the service sharing mechanisms and policies;
2) the user invocation sequence;
3) the service invocation sequence.

Appropriate internal service specifications are intended to eliminate undesirable interuser dependencies that may result from the first concern. Appropriate user agreement specifications are used to eliminate undesirable interuser dependencies that may result from the second and the third concerns. The service-specification model separates sharing mechanisms from policy specifications because it distinguishes different *types of service properties* (i.e., safety vs. liveness properties). We adopt a *temporal-logic-based* specification language ([Pnueli77, 79, 80], [Owicki82]) to facilitate expressing the semantics of sharing mechanisms and sharing policies within a service and the user agreements for this service. (The semantics of temporal logic are reviewed in Appendix 1.) However, other specification languages, not necessarily based on temporal logic, could be used with our specification method.

In the remainder of this section we explain how a specification that guarantees finite waiting for a service is written. We also explain the rationale for, and the specification of, the Finite Waiting Time (FWT) policy. Then we introduce a formal specification for shared services that can be invoked concurrently and a specification of user agreements that describes properties for safe user-invocation sequences and live service-invocation sequences. Finally, the relationship between different specifications that constitute the FWT policy is presented. We also discuss their progress implications.

4.4.1. Specification of the FWT Policy

Access to shared services must be guaranteed to authorized users. Thus, a FWT policy must be adopted in the service model to guarantee individual user progress within a shared service. For the purposes of this paper, the FWT policy consists of three different, yet mutually-related, specifications: *fairness* policies, *simultaneity* policies, and *user agreements*. The notion of the user agreement has been introduced earlier in this paper. The fairness and simultaneity policies are sharing policies specified within a service. The formal semantics of the fairness and simultaneity policies is presented in the next sub-section, which also introduces the formal specification of shared services. Informally, the fairness policy states that a user will not be blocked forever within a service if that user has many opportunities to make progress. The simultaneity policy states that a user will eventually have all the opportunities needed to make progress within a service provided that the user agreement of that service can be satisfied.

Whenever a specification implies individual user progress within a service, then the FWT policy for that service is guaranteed. Thus, if the user agreements for that service can be satisfied, then the simultaneity policies guarantee the existence of progress opportunities for each user, and the fairness policies, in turn, guarantee that each user makes progress. The user agreements of a service can be satisfied in one or both of the following two ways depending on the service environment:

1) apply constraints to all service users so that they obey the user agreements, and
2) provide facilities that enforce the user agreements so that sequences of user invocations are regulated before the actual calls are issued.

Therefore, the fairness policy and the simultaneity policy plus the user agreements satisfy the FWT policy.

Conversely, if any one of the specifications for fairness and simultaneity policies as well as the user agreements is not provided, then either the progress opportunities for some users may not always exist or the service treats some users unfairly. Thus, the enforcement of the FWT policy cannot be guaranteed.

A FWT policy is best specified by using, whenever possible, internal service specifications; i.e., sharing policy specifications. However, in general, it is impossible for internal service specifications to guarantee finite waiting times. This assertion is based on the following line of reasoning. First, FWT for a service is guaranteed only if the service invocation sequences of that service are live. However, a service specification cannot include the semantics of "live service invocation sequences" because it cannot predict users' behavior outside the service. Thus, we have to specify the properties required for live service-invocation sequences *outside* a service. Second, if we combined the fairness and the simultaneity policies into a single policy, then the resulting specification would not separate concerns properly and, in general, would become more complex. Such specification would convey too many properties and would become less comprehensible. This would make it difficult to implement the service from such a specification. For example, the fairness policy for users waiting within one entry queue (e.g., the FIFO policy), and the simultaneity policy (e.g., any policy that prevents individual user starvation) are implemented by different liveness properties. Any specification

that combines different, unrelated properties at the same level of abstraction would provide too little specific information for practical implementation of any of the properties. For this reason, we decompose the FWT specification into different types of specifications.

4.4.2. Service Specification

A service specification defines the properties for concurrent service access, and the necessary mechanisms and policies to enforce the desired properties. The skeleton of a service specification is given in Figure 2. Note the separation of the sharing mechanisms and policies. Below, we explain each keyword in Figure 2.

- **service:** This keyword gives the name of the service.
- **type:** The data types that will be used in the service are specified. Some well-known types such as integer, boolean are not specified here. All specified types include a type name. If the type name is self-evident, no further explanation is given. If a given type is a construct of other types, it is specified explicitly. For example:

  type *userid*
  units
  $index = 1 \ldots N$

- **constant:** The names of constant values or a group of structured constant values (e.g., constant array) are specified within the service. An example of constant specification is:

  constant *size* : *units*
  quota : **array** [*userid*] of *units*, $\forall id$: *userid*, $quota[id] \leq size$

  Note that constant specifications may be exported through the interface specifications.

- **variable:** The names of variables are specified. Variables are used to express the states of a service, such as the state of the shared resource, the number of concurrent users, etc. A variable is specified by its name, type, and initial value, such as

  variable *free* : *units*, **initially** *size*
  own : **array**[*userid*] of *units*, **initially** $\forall id$: *userid*, $own[id] = 0$

- **hidden operations:** The hidden operations are those which are not visible to the users outside the service. Therefore, users cannot invoke hidden operations. Hidden operations can only be referenced from within the service. In general, they are used to help describe the behavior of internal operations. The construct of an hidden operation includes an operation name and the effects of the operation.
- **interface operations:** A user can access a service only by invoking the interface operations. An interface operation may include arguments. The identity of the invoking user is assumed to be an implicit argument inherent in every interface operation. The argument of an operation "op" is expressed

Service Specification

service service_name

type

constant

variable

sharing mechanisms

hidden operations

operation_name $(arg_1, arg_2, \ldots, arg_n)$

effects :

interface operations

operation_name $(arg_1, arg_2, \ldots, arg_n)$

exception conditions :

effects :

resource constraints

concurrency constraints

sharing policies

fairness

simultaneity

Fig. 2. Service Specification Skeleton.

as the construct: *op.argument*. As an example of the resource allocator, *Acquire.id* represents the identity of the user process which invokes the "Acquire" operation currently. The construct of an interface operation includes an operation name (and arguments), exception conditions, and "effects" of the operation. Exception conditions describe the conditions under which the arguments will cause errors. The "effects" part describes the normal actions taken by the service when no exceptions occur. The variables within a service may change values after the execution of an interface operation. In order to distinguish the value of a service variable before modification from its value after modification, we add a symbol " ′ " to the right of the variable to represent its value after the operation has executed. For example, the "Acquire" operation of the resource allocator can be specified as follows, where n is the number of resources requested.

interface operations

Acquire (n : *units*)

exception conditions : $quota[id] < own[id] + n$

effects : $free' = free - n$

$id.own' = id.own + n$

- **resource constraints:** The specification of "resource constraints" is essentially the same as that of "resource invariance", which has been widely used in the literature [Hoare74], [Owicki79], [Hailpern82], etc. Resource constraints give the properties of the service-provided resource and should always be true for all service states. For example, in the resource allocation service, one of the resource constraints is that the number of resource units in the resource pool is always greater than or equal to zero and less than or equal to the maximum number of resource units of the resource pool:

resource constraints

$\Box((free \geq 0) \wedge (free \leq size))$

- **concurrency constraints:** Concurrency constraints specify the conditions under which concurrent operations are allowed to execute within a service. To specify such constraints, we use "$\#Active$" denoting the number of concurrent active interface operations in the service and "$\#Active\_op$" denoting the number of concurrent active operations "op" in the service. An operation is said to be active if it is currently performing its computation for changing the resource state. For examples, if mutual exclusion is required between interface operations for changing the resource state, and if an active operation will eventually terminate, then the concurrency constraints can be specified as follows:

concurrency constraints

1. $\Box(\#Active \leq 1)$
2. $(\#Active = 1) \rightsquigarrow (\#Active = 0)$

- **fairness:** The *Fairness* policy expresses the behavior required of a service such that no operations which

satisfy the necessary conditions *infinitely often* will be blocked forever. For example, in the resource allocation service, let *condition* c_1 denote the statement: "the number of resources currently available is no less than the number of resources requested", and let *condition* c_2 denote the statement: "there are no active operations in the service at this time". We need the following fairness policy: if both c_1 and c_2 can simultaneously become true infinitely often then the blocked "Acquire" operation will eventually resume execution and finally terminate. This *fairness* policy can be specified as follows:

fairness $\Big(at(\text{Acquire}) \wedge \Box\Diamond\left((free \geq Acquire.n) \wedge (\#Active = 0)\right)\Big) \rightsquigarrow after(\text{Acquire})$

where variable *free* is the number of currently available resources, and n is the argument to "Acquire" specifying the number of resources requested.

- **simultaneity:** The simultaneity policy states that during the waiting period of an invocation, if every *condition* requested can be satisfied infinitely often, then all of *conditions* eventually will be satisfied simultaneously. For example, let c_1 and c_2 be the same *conditions* as those used in the description of fairness policy. The simultaneity policy of the resource allocation service can be described as following two parts:

  1) whenever an invocation to "Acquire" is blocked, if c_1 can be satisfied infinitely often and so does c_2 then c_1 and c_2 eventually will be satisfied simultaneously.
  2) whenever an invocation to "Acquire" is blocked and some users always repeatedly release their allocated resources until c_1 becomes true then c_1 will eventually become true.

  In the second part, *condition* c_1 is further decomposed into a number of sub-conditions. Each of these sub-conditions denotes the statement: "one unit of requested resource become available." Thus, the number of sub-conditions to be fulfilled is the number of resources requested. We specify the simultaneity policy as follows:

  simultaneity

  1. $\Big(in(\text{Acquire}) \wedge \left(\Box\Diamond(free \geq Acquire.n)\right) \wedge \left(\Box\Diamond(\#Active = 0)\right)\Big)$
     $\rightsquigarrow \left((free \geq Acquite.n) \wedge (\#Active = 0)\right)$
  2. $\left(in(\text{Acquire}) \wedge \Box\Diamond(\#Active\_Release > 0)\right) \rightsquigarrow (free \geq Acquire.n)$

Agreement Specification

user agreement service_name

type

variable

internal agreement

safety

liveness

external agreement

safety

liveness

Fig. 3. Agreement Specification Skeleton.

4.4.3. Agreement Specification

As mentioned before, user agreements specify the properties that must be provided by the users of the shared services to prevent instances of denial of service. Some of the properties describe the actions the users are allowed to perform. These are *safety* properties. The others describe those actions which the users are required to perform. These are *liveness* properties. The skeleton of the agreement specification is shown in Figure 3.

- **internal agreement:** Internal agreements are required when a service includes *hidden* operations that can only be called from within that service. They specify the safety properties, such as "when a hidden operation is allowed to call," and/or the liveness properties, such as "when a hidden operation must be called." The internal operation execution sequences that contain hidden operations must satisfied the properties of the internal agreements. For example, the "two-phase locking" protocol [Gray 78], which is used to serialize user transactions in a distributed file access service, is a special form of internal agreement:

  safety $\forall u : \mathit{userid}, \forall f : \mathit{fileid}, \mathit{in}(u) \Rightarrow$

  1. $\neg \mathit{at}(\text{Read}(f))$ UNTIL $\mathit{after}(\text{Lock}(f, R))$
  2. $\neg \mathit{at}(\text{Write}(f))$ UNTIL $\mathit{after}(\text{Lock}(f, W))$
  3. $\mathit{at}(\text{Unlock}) \Rightarrow \Box\neg \mathit{at}(\text{Lock})$
  4. $\mathit{after}(\text{Lock}) \Rightarrow (\neg \mathit{after}(\text{Unlock})$ UNTIL $\mathit{at}(\text{Commit}))$

  liveness $\forall u : \mathit{userid}, \forall f : \mathit{fileid}, \mathit{in}(u) \Rightarrow$

  1. $\mathit{after}(\text{Lock}(f, R)) \rightsquigarrow \mathit{at}(\text{Read}(f))$
  2. $\mathit{after}(\text{Lock}(f, W)) \rightsquigarrow \mathit{at}(\text{Write}(f))$
  3. $\mathit{after}(\text{Lock}(f, \ )) \rightsquigarrow \mathit{at}(\text{Unlock}(f))$

In this internal agreement specification, "Read" and "Write" are interface operations, and "Lock" and "Unlock" are hidden operations. "Read(f)" means to read a file with file name "f", etc. "Lock(f, R)" means to request a Read lock on file "f". "Lock(f, W)" means to request a Write lock on file "f". If the "R" or "W" is not specified in a "Lock" operation, it could be a Read lock or a Write lock.

- **external agreement:** External agreements specify the allowed user operation order and the required service invocation sequence that must be ensured by users outside the service. As an example of a resource allocator, in order to guarantee that all users eventually can make progress, service invocation sequences must preserve a *liveness* property. Whenever an "Acquire" operation gets blocked waiting for some resources to become available, then a sufficient number of "Release" operations must *eventually* become active and finally terminate until sufficient free resource units become available. This allows the waiting "Acquire" operation to have a chance to make *progress*. This user agreement can be specified as:

external agreement

liveness

$$in(\text{Acquire}) \rightsquigarrow \Big((\Box\Diamond (\#Active\_Release > 0)) \vee (free \geq Acquire.n) \Big)$$

The user agreement in the service model is given in an abstract form and is valid for all services of the same class. However, the possible implementation of user agreements is strongly dependent on the service environment and thus may differ from each other. In the case of resource allocator, to guarantee the service invocation sequence to be live, users that share a class of resource allocation service may be restricted on some allowed operation orders such as the "ordered resource" approach [Havender 68]. Alternatively, service users can be asked to claim the largest number of resource units of each service that the user will need at one time before any service access, such as the "resource claim" approach [Habermann 69].

4.4.4. Progress Implications of Fairness, Simultaneity, and User Agreements

As mentioned earlier in this paper, the FWT is constructed from the fairness policies, the simultaneity policies, and the user agreements. After giving the semantics of these properties, we can now present more detailed discussion about the relationship between the fairness policy, the simultaneity policy, and user agreement. We also discuss their progress implications.

Informally, a fairness policy states that a user will make progress if it has many opportunities to make progress. A fairness policy is necessary to allow individual user progress. Without a fairness policy, an invocation may be blocked forever in the service even if it has many opportunities to make progress. To illustrate the notion of fairness, let us consider the most popular scheduling policy, the first-in-first-out policy (FIFO). Is FIFO a fair policy? First, if one defines FIFO according to the overall arrival order of service invocations, then FIFO is clearly not a fair policy. For example, some invocations of the "Release" operation of a resource allocator may be blocked forever just because they arrived after a call to "Acquire" that requested a number of resources which exceeded that of the currently available resources. Thus, invocations of "Release" are treated unfairly. Second, if one defines FIFO based on invocations of individual interface operations, then FIFO guarantees fairness only for invocations of the same service operation. It would not guarantee fairness between different service operations.

An important question is whether a fairness policy specification guarantees individual user progress. Fairness does not necessarily imply individual user progress because the opportunities for making progress may not always exist. Progress opportunities can only be provided by the application of simultaneity policy. Informally, the simultaneity policy states that a user will eventually have all the opportunities it needs to make progress if the user agreements allow these opportunities to occur. However, a fairness policy together with a simultaneity policy still cannot guarantee individual user progress because the existence of such opportunities always depends on user agreements.

To illustrate the necessity for user agreements, let us consider the specification of a fairness policy. The conditions required for progress within a service are expressed as the *hypotheses* of the fairness policy. Some of these conditions can be satisfied only when the service invocation sequence is live. Since shared services cannot predict users' behavior outside the service, the service specification is unable to include the semantics of live service-invocation sequences. Thus, in addition to specific sharing policies, some form of agreement specification is always required.

One may ask: if user agreements allow progress opportunities to be created, do we still need a simultaneity policy? The answer is yes. A simultaneity policy is required because, although user agreements can allow progress opportunities to be created, they may not be able to guarantee that progress takes place within a service for at least two reasons. First, the occurrence of progress opportunities may also depend on other conditions that can be satisfied only by sharing mechanisms and policies of the service itself. For example, a service may require mutual exclusion among concurrent service invocations. An operation invocation has

an opportunity to make progress only when the currently-active operation terminates. Termination would depend, in this case, upon the mutual exclusion conditions enforced *within* the service. Thus, the satisfaction of these conditions cannot be determined by the users outside the service. Second, even if individual conditions requested are all dependent on user agreements, they may not be satisfied simultaneously. This is because other invocations, which are required to satisfy only parts of these conditions, can make progress repeatedly. Thus, individual conditions requested by certain users may never become true simultaneously. The service invocations of these users will be blocked forever. The individual process starvation problem [Dijkstra 72] is an example of such a situation.

From the above discussion, it should be clear that the fairness policy, the simultaneity policy, and the user agreements are all required to guarantee individual progress. Of course, individual progress must also be supported by the application of appropriate sharing mechanisms and policies because the liveness property is dependent on safety properties of sharing mechanisms [Lamport 86].

4.5. APPLICATION OF SHARED SERVICE MODEL

In this sub-section, we apply the model of shared service to specify a general resource allocator. In order to demonstrate that the service model is appropriate for prevention of denial of service, we will formally verify that all users of the given resource allocator eventually make progress and receive intended service.

4.5.1. Specification of a Resource Allocator

One of the main shared services of an operating system is the allocation of system resources to users. A resource allocator consists of a pool of resource units that can be shared by a group of users. Initially, the pool contains the total number of resource units. To prevent resource monopolization, the resource allocator maintains a resource quota for each user. Resource quota provides the maximum number of resource units that can be assigned to each user. The resource allocator also maintains a variable array "*own*" that specifies the number of resource units currently assigned to each user. Each user can acquire "*n*" units of resource by invoking the "Acquire" operation. Similarly, each user can relinquish "*n*" units of resource by invoking the "Release" operation. Based on the model of shared service, the service and agreement specifications of the resource allocator are shown in Figures 4 and 5 respectively.

The agreement specification of the resource allocator does not include internal agreements because there are no *hidden* operations specified in the service specification of resource allocator.

Service Specification

service resource_allocator
type *userid*, *units*
constant $size$: *units*
$quota$: **array**[*userid*] of *units*, $\forall id$: *userid*, $quota[id] \leq size$
variable $free$: *units*, **initially** *size*
own : **array**[*userid*] of *units*, **initially** $\forall id$: *userid*, $own[id] = 0$
sharing mechanisms
interface operations
Acquire (n : *units*)
exception conditions : $quota[id] < own[id] + n$
effects : $free' = free - n$
$own[id]' = own[id] + n$
Release (n : *units*)
exception conditions : $n > own[id]$
effects : $free' = free + n$
$own[id]' = own[id] - n$
resource constraints
1. $\Box((free \geq 0) \wedge (free \leq size))$
2. $\forall id\ \Box((own[id] \geq 0) \wedge (own[id] \leq quota[id]))$
3. $(free = N) \Rightarrow \Big((free = N)\ \text{UNTIL}\ (after(\text{Acquire}) \vee after(\text{Release}))\Big)$
4. $\forall id\ (own[id] = M) \Rightarrow$
$\Big((own[id] = M)\ \text{UNTIL}\ (after(\text{Acquire}) \vee after(\text{Release}))\Big)$
concurrency constraints
1. $\Box(\#Active \leq 1)$
2. $(\#Active = 1) \rightsquigarrow (\#Active = 0)$
sharing policies
fairness
1. $\Big(at(\text{Acquire}) \wedge \Box\Diamond((free \geq Acquire.n) \wedge (\#Active = 0))\Big) \rightsquigarrow after(\text{Acquire})$
2. $\big(at(\text{Release}) \wedge \Box\Diamond(\#Active = 0)\big) \rightsquigarrow after(\text{Release})$
simultaneity
1. $\Big(in(\text{Acquire}) \wedge (\Box\Diamond(free \geq Acquire.n)) \wedge (\Box\Diamond(\#Active = 0))\Big)$
$\rightsquigarrow ((free \geq Acquite.n) \wedge (\#Active = 0))$
2. $\big(in(\text{Acquire}) \wedge \Box\Diamond(\#Active\_Release > 0)\big) \rightsquigarrow (free \geq Acquire.n)$

Fig. 4. Service Specification for the Resource Allocator

Agreement Specification

user agreement resource_allocator

external agreement

liveness

$$in(\text{Acquire}) \rightsquigarrow \Big((\Box\Diamond(\#Active\_Release > 0)) \vee (free \geq Acquire.n) \Big)$$

Fig. 5. Agreement Specification for the Resource Allocator

4.5.2. Formal Verification of the Resource Allocator

A resource allocator is prevented from denial of service if all user invocations which do not cause exceptions eventually terminate and receive their intended service. We begin by proving that the specifications of resource allocator guarantee that user invocations eventually make progress. We next show that the specifications ensure correct service for individual users. In the following proofs we assume that no user operations cause exceptions.

4.5.2.1. Progress Proofs

The term "invocations of resource allocator eventually terminate" can be expressed by the following two temporal formulas:

$$at(\text{Acquire}) \rightsquigarrow after(\text{Acquire}); \qquad (P1)$$

$$at(\text{Release}) \rightsquigarrow after(\text{Release}). \qquad (P2)$$

We must prove that given the service and agreement specifications of the resource allocator, both (P1) and (P2) are temporal theorems. We will prove a series of lemmas based on the service and agreement specifications. To prove that (P2) is a temporal theorem, we first show that the resource allocator is repeatedly in a state that no operations are in execution (Lemma 1). To prove that (P2) is a temporal theorem first, we show that a blocked "Acquire" invocation will eventually get a chance to proceed (Lemma 2). Second, we show that if an "Acquire" invocation can be blocked forever then it should have infinitely many chances to proceed (Lemma 3). Finally, we show that if $(P1)$ is not true then eventually the invocation will be blocked forever (Lemma 4). A list of derived temporal theorems, which are used in the following proofs, are given in the Appendix 2 for reference.

Lemma 1. The formula $\Box\Diamond(\#Active = 0)$ *is a temporal theorem.*

Proof: The concurrent constraints of the resource allocator imply

$$\Box(\neg(\#Active = 0) \Rightarrow \Diamond(\#Active = 0))$$

From the derived theorem $(D1)$ in the Appendix 2, we can conclude

$$\Box\Diamond(\#Active = 0)$$

Theorem 1. The formula $at(\text{Release}) \rightsquigarrow after(\text{Release})$ *is a temporal theorem.*

Proof: Applying *Lemma 1* to the second fairness policy of the resource allocator, we complete the proof immediately.

Lemma 2. The formula $in(\text{Acquire}) \rightsquigarrow (free \geq Acquire.n)$ *is a temporal theorem.*

Proof: From the user agreement specification of the resource allocator and the temporal axiom $(A5)$ in the Appendix 2, we have

$$\Box\big(in(\text{Acquire}) \Rightarrow \Diamond\big((\Box\Diamond(\#Active\_Release > 0)) \vee (free \geq Acquire.n)\big)\big).$$

From the derived theorem $(T3')$ in the Appendix 2, we have

$$\Box\big(in(\text{Acquire}) \Rightarrow \big((\Diamond\Box\Diamond(\#Active\_Release > 0)) \vee \Diamond(free \geq Acquire.n)\big)\big).$$

From the derived theorem $(T2')$ in the Appendix 2, we have

$$\Box\big(in(\text{Acquire}) \Rightarrow \big((\Box\Diamond(\#Active\_Release > 0)) \vee \Diamond(free \geq Acquire.n)\big)\big). \qquad (L1)$$

From the simultaneity policy 2 of the resource allocator and the temporal axiom $(A5)$ in the Appendix 2, we have

$\Box\big((in(\text{Acquire}) \wedge \Box\Diamond(\#Active\_Release > 0)) \Rightarrow \Diamond(free \geq Acquire.n)\big)$. Hence

$$\Box\big(\big(in(\text{Acquire}) \wedge ((\Box\Diamond(\#Active\_Release > 0)) \vee \Diamond(free \geq Acquire.n))\big) \Rightarrow \Diamond(free \geq Acquire.n)\big). \qquad (L2)$$

From the temporal formulas $(L1)$, $(L2)$ and the derived theorem $(D3)$ in the Appendix 2, we obtain

$$\Box\big(in(\text{Acquire}) \Rightarrow \Diamond(free \geq Acquire.n)\big).$$

Thus, we complete the proof.

Lemma 3. The formula $\Diamond\Box in(\text{Acquire}) \Rightarrow \Box\Diamond\big((free \geq Acquire.n) \wedge (\#Active = 0)\big)$ *is a temporal theorem.*

Proof: From *Lemma 1*, the first simultaneity policy of the resource allocator implies

$$\big(in(\text{Acquire}) \wedge \Box\Diamond(free \geq Acquire.n)\big) \rightsquigarrow \big((free \geq Acquire.n) \wedge (\#Active = 0)\big).$$

From the derived theorem $(D4)$ in the Appendix 2, we have

$$\big(\Diamond\Box\big(in(\text{Acquire}) \wedge \Box\Diamond(free \geq Acquire.n)\big)\big) \Rightarrow \Box\Diamond\big((free \geq Acquire.n) \wedge (\#Active = 0)\big).$$

From the derived theorems $(T4)$, $(T1)$ and $(T2')$ in the Appendix 2, we have

$$\big((\diamond\Box in(\text{Acquire})) \wedge \Box\diamond (free \geq Acquire.n)\big) \Rightarrow \Box\diamond \big((free \geq Acquire.n) \wedge (\#Active = 0)\big). \qquad (L3)$$

Also, *Lemma 2* implies

$$\diamond\Box\big(in(\text{Acquire})\big) \Rightarrow \Box\diamond (free \geq Acquire.n). \qquad (L4)$$

From the temporal formulas $(L3)$, $(L4)$ and the derived theorem $(D2)$ in the Appendix 2, we can conclude

$$\diamond\Box in(\text{Acquire}) \Rightarrow \Box\diamond \big((free \geq Acquire.n) \wedge (\#Active = 0)\big).$$

Lemma 4. Suppose that the temporal formula

$$\diamond\big(at(\text{Acquire}) \wedge \Box(\neg(after(\text{Acquire})))\big)$$

is true then

$$\diamond\Box\big(in(\text{Acquire})\big).$$

Proof: From the semantics of the control predicates *at, after* and *in* given in the Appendix 1, we have

$$\begin{aligned} &\diamond \big(at(\text{Acquire}) \wedge \Box(\neg(after(\text{Acquire})))\big) \\ \Rightarrow &\diamond \big(at(\text{Acquire}) \wedge \Box(in(\text{Acquire}))\big) \\ \Rightarrow &\diamond \Box\big(in(\text{Acquire})\big). \end{aligned}$$

Therefore, we complete the proof.

Theorem 2. The formula $at(\text{Acquire}) \rightsquigarrow after(\text{Acquire})$ *is a temporal theorem.*

Proof: Suppose it is not a temporal theorem then

$$\diamond\big(at(\text{Acquire}) \wedge \Box(\neg(after(\text{Acquire})))\big)$$

From Lemma 3 and 4 we obtain

$$\Box\diamond \big((free \geq Acquire.n) \wedge (\#Active = 0)\big)$$

Then, from the first fairness policy, we obtain $at(\text{Acquire}) \rightsquigarrow after(\text{Acquire})$ which is supposed to be false, a contradiction occurs. Thus, we can conclude the theorem.

4.5.2.2. Correct Service Proofs

To demonstrate that user invocations receive correct service when they terminate, we show that the following temporal formulas are temporal theorems.

$$\forall id\big((own[id] = M) \wedge \diamond(after(\text{Acquire}(n)))\big) \Rightarrow \diamond(own[id] = M + n), \qquad (C1)$$

$$\forall id\big((own[id] = M) \wedge \diamond(after(\text{Release}(n)))\big) \Rightarrow \diamond(own[id] = M - n), \qquad (C2)$$

$$\forall id\big((own[id] = M) \wedge \Box(\neg(after(\text{Acquire})) \wedge \neg(after(\text{Release})))\big)$$
$$\Rightarrow \Box(own[id] = M). \qquad (C3)$$

The proofs are rather straightforward by applying the specification of sharing mechanisms.

Theorem 3. The temporal formulas (C1), (C2), and (C3) are theorems.

Proof. First, from the concurrency constraints of resource allocator, concurrent invocations are required to be executed mutual exclusively. Thus, if $\diamond(after(\text{Acquire}(n)))$ then the "effects" part of "Acquire" operation implies (C1) directly. Similarly, the "effects" part of "Release" operation implies (C2) directly.

Second, applying the temporal axiom $(P \text{ UNTIL } Q) \Rightarrow (\Box\neg Q \Rightarrow \Box P)$, (C3) is directly implied by the fourth resource constraint.

4.6. DISCUSSION

We have presented here a formal service model for the prevention of denial of service in computer systems. We have chosen a specification method based on temporal logic to facilitate the construction of a service model for two reasons: 1) it has the power of reasoning about "future" events, and thus is particularly suitable for expressing our notion of progress; 2) it is convenient for expressing the semantics of the invocation order, and this order relation is an important part of our service model. The advantage of using this specification method to solve safety and liveness problems is that it states the essential service properties based on the concept of *abstraction*. The abstract service specification model must be interpreted differently in different service environments. Thus this model is appropriate for the prevention of denial-of-service problem because the problem appears in various guises in practice [Gligor 83].

Other research work also used abstract specification methods to describe required properties [Owicki 82], [Hailpern 82], [Lamport 83], [Ramamritham 85]. However, these methods are less suitable for the prevention of denial-of-service problems for the following reasons:

- they do not provide formal specifications of simultaneity policy within a service. (Although in principle these methods *could* provide simultaneity policy specifications, they currently do not include such specifications.)
- they do not specify properties that must be satisfied by users *outside* the service. (Instead, such properties usually appear as hypotheses of fairness policies specified within a service. However, user behavior

outside the service may make these hypotheses false. Therefore, external service specifications are still necessary.)

To date, only fairness policies have been formally specified and verified in a general way. However, a fairness policy cannot prevent denial-of-service instances caused by conspiracy among a group of users that manages to monopolize shared resources. A solution for the prevention of individual process starvation in a specific service has been proposed informally for the dining philosophers problem [Dijkstra 72] based on a characterization of simultaneity policies. Our service specification and verification method provides formal semantics for simultaneity policies that are applicable to all shared services. Simultaneity policies are necessary but are not sufficient to guarantee individual user progress within a service.

The main reason the current specification methods are unsatisfactory is that they only attempt to express the properties that must be enforced *within* a service. User agreement specifications, which are necessary to eliminate the effects of undesirable interuser dependencies within the service, are conspicuous by their absence. Therefore, denial-of-service problems cannot be solved by direct application of current specification methods.

5. FORMAL IMPLEMENTATION AND VERIFICATION OF ADA SERVICES

5.1. SERVICES IN THE PROGRAMMING LANGUAGE ADA

Ada is a language that addresses many important programming issues including strong typing, data abstraction, tasking, generic units and exception handling, etc. Ada is an adequate language for the implementation of shared services due to its strong expressive power. For example, the Ada tasking facility provides synchronization mechanisms and the Ada packaging facility provides a suitable tool for service encapsulation [Wegner83]. The prevention of denial of service in Ada implementations of shared services is strongly dependent on the semantics of tasking and access types. In this section, we focus on these two issues. For a general view of Ada, the reader should refer to the Ada reference manual [Goos83].

5.1.1. Ada tasking

The basic synchronization and communication mechanism for Ada tasks is the *rendezvous* mechanism. The rendezvous mechanism of Ada represents the action of synchronization followed by communication. A user task may invoke an Ada service by issuing an entry call to the service task in which the rendezvous mechanism is provided. A rendezvous between the caller task and service task occurs whenever the caller invokes an entry call declared in the service task and the service task reaches the "accept" statement for the entry invoked by the caller. During the rendezvous, the caller task is suspended until the execution of the "accept body" is completed.

A user call to an Ada task will wait in the related entry queue before it can be accepted for service rendezvous. User calls within an entry queue are scheduled for service by the order of first-in-first-out (FIFO). When multiple open entries exist, Ada may select among them in an arbitrary manner. There is no guarantee of fair selection of open entries in Ada. Thus, to guarantee that each user makes progress, a fairness policy must be enforced within the service task whenever it schedules user calls among open entries.

The communication mechanism of Ada is asymmetric in that the service task need not know the identity of its caller. Thus, in general, the service tasks are structured as an endless loop to await open entries.

5.1.2. Access Type

In Ada, objects of the "access" type provide a means to access other objects and these other objects must be allocated dynamically from heaps. An access object can be considered to be a pointer to another object. Tasks can also be created through access types. Tasks created through access types are made active immediately on the evaluation of the allocator. Such tasks are not dependent on the unit where they are created. Instead, they are dependent on the unit containing the declaration of the access type itself. Thus, the lifetime of such a task is independent of the lifetime of its creator. This feature enables Ada to cope with some denial-of-service problems such as abrupt aborts of user tasks that cause data inconsistency within the service.

5.2. IMPLEMENTATION OF ADA SERVICES FROM SPECIFICATIONS

Based on the formal service specification and the semantics of Ada, we are able to construct Ada services that satisfy their formal specifications. Since the Ada tasking provides the rendezvous mechanism for synchronizing user tasks within the service task, the concurrency constraints of the service specification is inherently preserved within the Ada tasks. As mentioned before, a fair scheduling policy is inherent within an entry queue by the FIFO ordering. However, one needs to implement a fair selection policy among open entries. Except for these features, other properties that are needed to be implemented within the Ada service for the prevention of denial of service depend on the characteristics and environment of a specific service.

Below, we implement an Ada resource allocator service from its formal specification provided in the previous section. Two implementations of the resource allocator are given. The first is based on a strict FIFO order for resource allocation. The second is based on the availability of resources during service request and thus it does not necessary follow the FIFO order. Both resource allocator implementations include two parts: the service part, which is constructed from the service specification of resource allocator, and the user invocation part, which is constructed from the user agreement specification of the resource allocator.

5.2.1. The FIFO Resource Allocator

5.2.1.1. Implementation of the Service Task

The resource allocator with strict FIFO order on resource allocation explicitly ensures the fairness property of the service. Each user invocation will eventually be granted the service whenever all users obey appropriate user agreements. The FIFO policy is inherent within individual entry queues. A FIFO policy among open ACQUIRE calls can be ensured by not accepting other ACQUIRE entry calls for the next rendezvous if the current ACQUIRE entry call in the rendezvous has not received the requested resources. This is illustrated in the Ada task of FIFO resource allocator below:

```
task FIFO_RESOURCE is
    ERROR: exception;
    entry ACQUIRE(ID: in USERID; N: in UNITS);
    entry RELEASE(ID: in USERID; N: in UNITS);
end;

task body FIFO_RESOURCE is
    FREE: UNITS: = MAX;
    QUOTA: constant array (USERID range <>) of UNITS: = ( ··· );
    OWN: array (USERID range <>) of UNITS: = (OWN'RANGE => 0);
begin
    loop
        begin
            select
                accept ACQUIRE(ID: in USERID; N: in UNITS) do
                    if QUOTA(ID) < OWN(ID) + N then
                        raise QUOTA_ERROR;
                    end if;
                    while FREE < N loop
                        begin
                            accept RELEASE(RID: in USERID; M: in UNITS) do
                                if M > OWN(RID) then
                                    raise QUOTA_ERROR;
                                end if;
                                FREE: = FREE + M;
                                OWN(RID): = OWN(RID) − M;
                            end;
                        exception
                            when QUOTA_ERROR =>
                                null;
                        end;
                    end loop;
```

```
                    FREE: = FREE − N;
                    OWN(ID): = OWN(ID) + N;
                end;
            or
                accept RELEASE(ID: in USERID; N: in UNITS) do
                    if N > OWN(ID) then
                        raise QUOTA_ERROR;
                    end if;
                    FREE: = FREE + N;
                    OWN(ID): = OWN(ID) − N;
                end;
            end select;
        exception
            when QUOTA_ERROR =>
                null;
        end;
    end loop;
end FIFO_RESOURCE;
```

Here, the type USERID and UNITS are declared in a package containing the service task FIFO_RESOURCE. The USERID defines the type of user identity and the UNITS defines the type of resource unit. The task FIFO_RESOURCE provides two entries for the service. The ACQUIRE(N) and RELEASE(N) entries can be invoked by users to acquire and relinquish "N" units of resource respectively. FREE is a variable of type UNITS that represents the number of resource units available at any instance of time. The total number of resource units is MAX. QUOTA is a constant array with index of type USERID. Each component of QUOTA defines the maximum amount of resources that can be allocated to the index user. OWN is a variable array with index of type USERID. Each component of OWN contains the amount of resources currently allocated to the index user. If a user attempts to acquire resource exceeding its quota, or if it attempts to relinquish resources that are not allocated, then the service task raises a QUOTA_ERROR exception. The exception will propagate to both the service and the invoking task. The exception handlers within the service task ignore the request and allow the service task to carry on the next rendezvous. Note that the exception handler within the ACQUIRE operation is indispensable; otherwise, the current ACQUIRE operation in rendezvous will be blocked forever, and eventually the entire service task will become inactive.

5.2.1.2. The Implementation of User Agreements

The specification of the user agreements for the resource allocator requires that, whenever a user is waiting for some resources, some other users relinquish their allocated resources until enough resources are available. Based on this user agreement, user tasks can be implemented as follows:

```
task type USER;

task body USER is
    ID: USERID;
    N: UNITS;
begin
    -- other computations not involving ACQUIRE and RELEASE
    FIFO_RESOURCE.ACQUIRE(ID, N);
    -- work on the acquired resources with normal termination
    FIFO_RESOURCE.RELEASE(ID, N);
    -- other computations not involving ACQUIRE and RELEASE
exception
    when FIFO_RESOURCE.QUOTA_ERROR =>
        null;
    when others =>
        FIFO_RESOURCE.RELEASE(ID, N);
end USER;
```

The user task implemented above satisfies the user agreement specification because users are required to relinquish their acquired resources before the user task terminates. If the user task requests resources exceeding its quota, then it will receive a QUOTA_ERROR exception from the service task. The exception handler ignores the invocation and terminates the user task immediately. If the call to ACQUIRE does not cause exception, then neither does the following call to RELEASE because the RELEASE call just relinquishes the same amount of resource acquired. If any other exception occurs between the call to ACQUIRE and the call to RELEASE for any reason, then the exception handler relinquishes the allocated resources and terminates the user task immediately. If any other exception occurs before the call to ACQUIRE or after the call to RELEASE, then the exception handler also attempts to relinquish resources as though they have been allocated. This will not cause allocation problems because the service task will then raise a QUOTA_ERROR exception and ignore the call. When the user task receives this exception, it also ignores the exception and terminates.

However, this simple implementation is only feasible in an environment where user processes are well protected. In an environment, such as application level services, where abnormal termination of a user task is possible, the above implementation may not sufficient, because an abnormal abort of a user task may

cause data inconsistency within the service. For example, if the user task is aborted during the call to ACQUIRE during a rendezvous, then the requested resources will get lost because the service task will carry on the rendezvous to completion. Similarly, if the user task is aborted during the call to RELEASE before a rendezvous, then the allocated resources cannot be relinquished because the user task will be removed from the entry queue of the RELEASE operation.

One way to prevent these inconsistencies is to make a service invocation independent of its caller tasks so that the calls to the entry operations of the service task will not terminate abruptly even when the caller aborts. In Ada, the way to achieve such invocation independence is through the use of the access type introduced before. In the resource allocator, task types ACQ_AGENT and REL_AGENT can be declared within a certain package body that contains the task FIFO_RESOURCE. These task types constitute the agent that actually issues calls to the entries of service task FIFO_RESOURCE.

The service package FIFO_RESOURCE_ALLOCATOR given below is implemented with the access-type agent in it. It contains procedures ACQUIRE and RELEASE, which serve as interfaces to the users outside the package. Every time the ACQUIRE procedure is called, it creates a new ACQ_AGENT task on behalf of the caller. Since the access type is declared within the package body of the resource allocator, it will not depend on the caller; thus, the created agent will not be aborted abnormally. If the user task is aborted during a call to ACQUIRE, the created ACQ_AGENT will return the acquired resources back to the resource pool of the service task. Therefore, the use of ACQ_AGENT prevents the loss of resources that would occur if user tasks could call the ACQUIRE operation of service task FIFO_RESOURCE directly.

The ACQUIRE procedure also creates a REL_AGENT before termination. The identity of this agent (i.e., a pointer) is returned to the the user that called the ACQUIRE procedure. The agent identity is declared as a private type within the package, thus the value of the agent identity cannot be changed by the users outside the package although its value can be transferred among user tasks. A user must specify the agent identity as an argument whenever it invokes the RELEASE procedure. The agent identity enables the user to release the resources by using the correct agent. This facility guarantees that users can only relinquish resources they are authorized to release. When the user task is aborted during a call to RELEASE, the REL_AGENT will not terminate abnormally. Thus, the resources intended to be released can be returned to the resource pool of the service task successfully. Without access-type agent REL_AGENT, the intended resource release cannot be guaranteed.

Note that the declaration of QUOTA_ERROR exception is moved from the service task FIFO_RESOURCE to the package FIFO_RESOURCE_ALLOCATOR. This change enables user tasks to receive the exception from the package directly.

```
generic
    MAX: NATURAL;
    type USERID is (<>);
```

```
package FIFO_RESOURCE_ALLOCATOR is
    QUOTA_ERROR: exception;
    type REF_ACQUIRE is limited private;
    type REF_RELEASE is limited private;
    type UNITS is private;
    procedure ACQUIRE(ID: in USERID; R: out REF_RELEASE; N: in UNITS);
    procedure RELEASE(ID: in USERID; R: in REF_RELEASE; N: in UNITS);

private
    type UNITS is INTEGER range 0..MAX;
    task type ACQ_AGENT;
    task type REL_AGENT;
    type REF_ACQUIRE is access ACQ_AGENT;
    type REF_RELEASE is access REL_AGENT;
    task type ACQ_AGENT is
        entry ACQ(ID: in USERID; R: out REF_RELEASE; N: in UNITS);
    end;
    task type REL_AGENT is
        entry REL(ID: in USERID; R: in REF_RELEASE; N: in UNITS);
    end;
end;

package body FIFO_RESOURCE_ALLOCATOR is

    task FIFO_RESOURCE is
        entry ACQUIRE(ID: in USERID; N: in UNITS);
        entry RELEASE(ID: in USERID; N: in UNITS);
    end;

    task body FIFO_RESOURCE is

        -- as before

    end FIFO_RESOURCE;

    task body ACQ_AGENT is
    begin
        select
            accept ACQ(ID: in USERID; R: out REF_RELEASE; N: in UNITS) do
                FIFO_RESOURCE.ACQUIRE(ID, N);
                -- assume that the boolean variable ABORTED is true
                -- if and only if the calling task is aborted;
                if ABORTED then
                    FIFO_RESOURCE.RELEASE(ID, N);
                else
                    R: = new REL_AGENT;
                end if;
            end;
        or
```

```
            terminate;
        end select;
    end ACQ_AGENT;

    task body REL_AGENT is
    begin
        select
            accept REL(ID: in USERID; R: in REF_RELEASE; N: in UNITS) do
                FIFO_RESOURCE.RELEASE(ID, N);
            end;
        or
            terminate;
        end select;
    end REL_AGENT;

    procedure ACQUIRE(ID: in USERID; R: out REF_RELEASE; N: in UNITS) is
        A: REF_ACQUIRE: = new ACQ_AGENT;
    begin
        A.ACQ(ID, R, N);
    end ACQUIRE;

    procedure RELEASE(ID: in USERID; R: in REF_RELEASE; N: in UNITS) is
        R.REL(ID, R, N);
    end RELEASE;

end FIFO_RESOURCE_ALLOCATOR;
```

User tasks outside the package cannot call directly the entry operations of the service task FIFO_RESOURCE. Instead they can only access the service-provided resources by issuing calls to the package procedures. Thus, the Ada code of user task USER described before needs to be modified so that the name of the service task (i.e., FIFO_RESOURCE) referenced within the user task USER is replaced by the the name of the service package (i.e., FIFO_RESOURCE_ALLOCATOR) that encapsulates the service task. The identity of the agent, which actually invokes the service task on behalf of the package user, must also be provided as an argument whenever a package procedure is called.

In the new version of user tasks given below we assume no abortion occurs between the call to ACQUIRE and RELEASE. Otherwise, the system must detect the occurrence of the abortion and release the allocated resources on behalf of the aborted user task. This is true because the service package has no way to detect user task abortions other than those that occur during service invocations.

```
task type FIFO_PACKAGE_USER;

task body FIFO_PACKAGE_USER is
    ID: USERID;
    R: REF_RELEASE;
    N: UNITS;
```

```
begin
    -- other computations not involving ACQUIRE and RELEASE
    FIFO_RESOURCE_ALLOCATOR.ACQUIRE(ID, R, N);
    -- work on the acquired resources with normal termination
    FIFO_RESOURCE_ALLOCATOR.RELEASE(ID, R, N);
    -- other computations not involving ACQUIRE and RELEASE
exception
    when FIFO_RESOURCE_ALLOCATOR.QUOTA_ERROR =>
        null;
    when others =>
        FIFO_RESOURCE_ALLOCATOR.RELEASE(ID, R, N);
end FIFO_PACKAGE_USER;
```

5.2.2. The General Resource Allocator

5.2.2.1. Implementation of Service Task

A FIFO resource allocator, although easy to implement, is inefficient especially when some users request a large amount of resources. This is because the computation of other users many be delayed within the resource allocator for a long time. The general resource allocator will not use the FIFO policy, instead it will use a general policy that is based on the availability of resource during service request. A non-FIFO resource allocator may cause an unfairness problem. However, by explicitly setting a maximum waiting time for each user, individual users are guaranteed to make progress within the service.

Below we provide a non-FIFO task of resource allocator where the maximum waiting time for users is explicitly defined. Here the maximum waiting time is not expressed as calendar time. Instead it is expressed as the number of request tries before the request is accepted by the service task.

```
task RESOURCE is;
  QUOTA_ERROR: exception;
  entry RESERVE(ID: in USERID; N: in UNITS; OK: out BOOLEAN);
  entry TRY(ID: in USERID; N: in UNITS; R: in REF_WAIT; OK: out BOOLEAN);
  entry RELEASE(ID: in USERID; N: in UNITS);
end;

task body RESOURCE is
  FREE: UNITS: = MAX;
  QUOTA: constant array (USERID range <>) of UNITS: = ( ··· );
  OWN: array (USERID range <>) of UNITS:= (OWN'RANGE => 0);
begin
  loop
    begin
      select
        accept RESERVE(ID: in USERID; N: in UNITS; OK: out BOOLEAN) do
          if QUOTA(ID) < OWN(ID) + N then
            raise QUOTA_ERROR;
          end if;
          if N <= FREE then
            FREE: = FREE − N;
            OWN(ID): = OWN(ID) + N;
            OK: = TRUE;
          else
            OK: = FALSE;
          end if;
        end;
      or
        accept RELEASE(ID: in USERID; N: in UNITS) do
```

```
            if N > OWN(ID) then
              raise QUOTA_ERROR;
            end if;
          FREE: = FREE + N;
            OWN(ID): = OWN(ID) − N;
          end;
          for I in 1..TRY'COUNT loop
            accept TRY(ID: in USERID; N: in UNITS; R: in REF_WAIT; OK: out BOOLEAN) do
              while FREE < N and then R.all = MWT do
                begin
                  accept RELEASE(RID: in USERID; M: in UNITS) do;
                    if M > OWN(RID) then
                      raise QUOTA_ERROR;
                    end if;
                    FREE: = FREE + M;
                    OWN(RID): = OWN(RID) − M;
                  end;
                exception
                  when QUOTA_ERROR =>
                    null;
                end;
              end loop;
              if N <= FREE then
                FREE: = FREE − N;
                OWN(ID): = OWN(ID) + N;
                OK: = TRUE;
              else
                OK: = FALSE;
              end if;
            end;
          end loop;
        end select;
      exception
        when QUOTA_ERROR =>
          null;
      end;
    end loop;
  end RESOURCE;
```

The service task RESOURCE includes three entries, RESERVE, TRY, and RELEASE. A call to RESERVE or to RELEASE is always accepted assuming no exception occurs. If a call to RESERVE is unsatisfied (i.e., return with OK = FALSE), the user may request the service task again by invoking the TRY entry. After a call to RELEASE, the calls waiting on the TRY entry queue are considered since the resources relinquished by the RELEASE call may available for one or more waiting TRYs. The waiting queue is scanned

by doing a rendezvous with each TRY call. Whenever the number of TRY calls issued by a user reaches the predefined maximum waiting time, the scanning action is delayed until the current user request is satisfied. This is accomplished by accepting RELEASE calls repeatly until FREE $\geq$ N; this ensures that no TRY calls will be delayed forever.

We use a variable of type REF_WAIT to denote current number of TRY calls issued by the invoking user task. If user tasks are allowed to invoke entry operations of the service task RESOURCE directly, then they are responsible for counting the number of TRY calls; the service only checks whether the number of TRY calls has reached the preset maximum waiting time. The type REF_WAIT is defined below.

```
type REF_WAIT is access POS_INT range 0 .. MWT.
```

5.2.2.2. The Implementation of User Agreements

Users of service task RESOURCE are responsible for: (1) issuing a RELEASE call if and only if there exists a satisfied resource request; (2) issuing a RESERVE call only for the first resource request, and then calling TRY repeatly if the previous request is not satisfied; and (3) counting the number of TRY calls and providing input to the service using the TRY call. Based on these requirements, user tasks can be implemented as follows:

```
task type USER;
task body USER is
    ID: USERID;
    N: UNITS;
    PASSED: BOOLEAN;
    RWT: REF_WAIT: = new POS_INT(0);
begin
    -- other computations not involving ACQUIRE and RELEASE
    RESOURCE.RESERVE(ID, N, PASSED);
    while not PASSED loop
        RWT.all: = RWT.all + 1;
        RESOURCE.TRY(ID, N, RWT, PASSED);
    end loop;
    -- work on the acquired resources with normal termination
    RESOURCE.RELEASE(ID, N);
    -- other computations not involving ACQUIRE and RELEASE
exception
    when RESOURCE.QUOTA_ERROR =>
        null;
    when others =>
        RESOURCE.RELEASE(ID, N);
end USER;
```

For preventing abrupt termination during *rendezvous*, the above user task is inadequate for the same reasons as those given in the implementation of the user agreement of FIFO resource allocator (Section 5.2.1.2). In order to make the execution of invocations independent of the service user, access-type agents are still required. The task RESOURCE will be encapsulated in the Ada package RESOURCE_ALLOCATOR, which also declares agents ACQ_AGENT and REL_AGENT. The package has two procedures, ACQUIRE and RELEASE. The RELEASE procedure calls the RELEASE entry of task RESOURCE directly, whereas the ACQUIRE procedure encapsulates the RESERVE call, the retry calls, and the retry counting. The package RESOURCE_ALLOCATOR is shown below:

```
generic
    MAX: NATURAL;
    type USERID is (<>);
    subtype POS_INT is INTEGER range 0..INTEGER'LAST;
    MWT: POS_INT;

package RESOURCE_ALLOCATOR is
    QUOTA_ERROR: exception;
    type REF_ACQUIRE is limited private;
    type REF_RELEASE is limited private;
    type UNITS is private;
    procedure ACQUIRE(ID: in USERID; R: out REF_RELEASE; N: in UNITS);
    procedure RELEASE(ID: in USERID; R: in REF_RELEASE; N: in UNITS);

private
    type UNITS is INTEGER range 0..MAX;
    task type ACQ_AGENT;
    task type REL_AGENT;
    type REF_ACQUIRE is access ACQ_AGENT;
    type REF_RELEASE is access REL_AGENT;
    task type ACQ_AGENT is
        entry ACQ(ID: in USERID; R: out REF_RELEASE; N: in UNITS);
    end;
    task type REL_AGENT is
        entry REL(ID: in USERID; R: in REF_RELEASE; N: in UNITS);
    end;
    type REF_WAIT is access POS_INT range 0..MWT;
end;

package body RESOURCE_ALLOCATOR is

    task RESOURCE is;
        entry RESERVE(ID: in USERID; N: in UNITS; OK: out BOOLEAN);
        entry TRY(ID: in USERID; N: in UNITS; W: in REF_WAIT; OK: out BOOLEAN);
        entry RELEASE(ID: in USERID; N: in UNITS);
    end;

    task body RESOURCE is
```

```
        -- as before

    task body ACQ_AGENT is
        PASSED: BOOLEAN;
        RWT: REF_WAIT: = new POS_INT(0);
    begin
        select
            accept ACQ(ID: in USERID; R: out REF_RELEASE; N: in UNITS) do
                RESOURCE.RESERVE(ID, N, PASSED);
                while not PASSED loop
                    RWT.all: = RWT.all + 1;
                    RESOURCE.TRY(ID, N, RWT, PASSED);
                end loop;
                -- Assume that the boolean variable ABORTED is true
                -- if and only if the calling task is aborted.
                if ABORTED then
                    RESOURCE.RELEASE(ID, N);
                else
                    R: = new REL_AGENT;
                end if;
            end;
        or
            terminate;
        end select;
    end ACQ_AGENT;

    task body REL_AGENT is
    begin
        select
            accept REL(ID: in USERID; R: in REF_RELEASE; N: in UNITS) do
                RESOURCE.RELEASE(ID, N);
            end;
        or
            terminate;
        end select;
    end REL_AGENT;

    procedure ACQUIRE(ID: in USERID; R: out REF_RELEASE; N: in UNITS) is
        A: REF_ACQUIRE: = new ACQ_AGENT;
    begin
        A.ACQ(ID, R, N);
    end ACQUIRE;

    procedure RELEASE(ID: in USERID; R: in REF_RELEASE; N: in UNITS) is
        R.REL(ID, R, N);
    end RELEASE;

end RESOURCE_ALLOCATOR;
```

Given the package RESOURCE_ALLOCATOR, the user task of the package is:

```
task type PACKAGE_USER;

task body PACKAGE_USER is
    ID: USERID;
    R: REF_RELEASE;
    N: UNITS;
begin
    -- other computations not involving ACQUIRE and RELEASE
    RESOURCE_ALLOCATOR.ACQUIRE(ID, R, N);
    -- work on the acquired resources with normal termination
    RESOURCE_ALLOCATOR.RELEASE(ID, R, N);
    -- other computations not involving ACQUIRE and RELEASE
exception
    when RESOURCE_ALLOCATOR. QUOTA_ERROR =>
        null;
    when others =>
        RESOURCE_ALLOCATOR.RELEASE(ID, R, N);
end PACKAGE_USER;
```

5.3. FORMAL VERIFICATION OF ADA SERVICES

To show the correctness of preventing denial-of-service problems in Ada services, we need to verify that the Ada services preserve all the properties specified in the service and agreement specifications. Properties are expressed by assertions about service states. An assertion is a formula constructed from service entities and other derived formulas by using predefined formation rules. Entities of a service include types, constants, variables, operators, and operations. The verification of the Ada services is accomplished by three major steps:

1) For each entity e_s of the service specification we provide an entity e_a of the Ada service, called the representing entity for e_s in the Ada service.
2) For each formula S of the service specification we induce a formula A of Ada, called the representing formula for S in the Ada service, by substituting for individual entities of S their Ada representations.
3) For each representing formula A we verify its validity in the Ada service; i.e., we prove that the representing formula is a theorem in the Ada service.

This verification method is an inverse of general methods used for verifying the correctness of the implementation of an specification [Hoare72], [Wulf et al.76], [Walker80]. In these methods, a mapping function *map* is used to map any state α_i of implementation to the corresponding state α_s of specification. The current state is defined by the values of variables declared. By using valid assertions (e.g., invariants) of

implementation, the verification shows that for any specification formula S about the states of specification, S is true of $map(\alpha_i)$.

In contrast, the mapping function in this paper is established in the reverse way such that the state α_s of the specification is represented in the state α_i of the implementation. To accomplish this, a representation *rep* is used to map each entity from specification to implementation. Then, each specification formula S is represented in implementation by substituting for each S entity its implementation representation. The verification shows that for any representing formula A, A is true of α_i.

The notion of representation used here is not new. It has been used in the Ina Jo specification language [Scheid et al.86] and, informally, in [Gligor83a]. In the Berry' paper [Berry87], it is argued that the two verification methods mentioned above imply each other. That is, for any formula S of the specification and any state α_i of implementation, S is true of $map(\alpha_i)$, if and only if $rep(S)$ is true of α_i. Our verification method, however, will focus on the proofs of liveness assertions that are not provided in the Theorem Prover of the Ina Jo language, or of any other formal specification language.

5.3.1. Implications of Ada Service Verification

In our Ada services verification method, the verification step 3 may not be straightforward because one may not easy to obtain, at the first try, a correct and (functional) complete implementation from a given service specification. Thus, after obtaining an Ada implementation, if one finds that some of the representing formulas cannot be proven in the implementation, then the implementation is incorrect or incomplete. In practice, one may need to modify the original implementation several times before obtaining a correct and complete Ada service, an implementation in which all of the representing formulas can be proven.

The main difficulty of obtaining a correct and complete implementation comes from the limitations and constraints of individual implementation languages. Such restrictions are ignored in service specifications, which is supposed to be abstract, but must be taken into account for service implementation. For example, the Ada language requires that an argument-dependent *condition* to be checked after selecting an operation (with arguments) for rendezvous. Thus, an Ada service whose sharing policies depend on argument values must be implemented in a more roundabout manner [Liskov et al.86]. Such Ada limitations are also encountered in our Ada implementation of resource allocators. For example, the *condition* "availability of resources" has to be checked within the ACQUIRE operation instead of using a more convenient Ada facility, such as using a "when" clause outside the operation (i.e., before the "accept" clause). This is because the availability of resources depends on the number of units of resources required, which is an argument to the ACQUIRE operation call.

5.3.2. The Formal Representations

A. Entity Representation

Each entity (i.e., type, constant, variable, operator, and operation) must has a representation in Ada service. Suppose that e_a is the implementation representation for a specification entity e_s in Ada service. Then the mapping from e_s to e_a is related by the representation *rep* as follows:

$$rep(e_s) = e_a.$$

If an entity of specification has parameters, such as an operation, then its representing entity also has parameters. Suppose that the specification entity e_s has n parameters of types $t_1, \cdots, t_n$, then its representing entity $rep(e_s)$ also has n parameters of types $rep(t_1), \cdots, rep(t_n)$ that are already defined in the representation.

An unary operator is considered an entity with one parameter and a binary operator is an entity with two parameters. For example, the expression of the logical "and" operator $\wedge(e_1, e_2)$ is equivalent to the usual form $e_1 \wedge e_2$. Temporal operators are presumed to represented by themselves and thus, $\Box, \Diamond$, UNTIL, $\rightsquigarrow$ map to $\Box, \Diamond$, UNTIL, $\rightsquigarrow$ respectively.

B. Formula Representation

Formulas are constructed from service entities. Let $S(e_{s1}, \cdots, e_{sn})$ be a specification formula. Suppose $rep(e_{s1}) = e_{a1}, \cdots, rep(e_{sn}) = e_{an}$, where $e_{s1}, \cdots, e_{sn}$ are entities of service specification and $e_{a1}, \cdots, e_{an}$ are entities of Ada services, we define

$rep\big(S(e_{s1}, \cdots, e_{sn})\big) = A(e_{a1}, \cdots, e_{an})$ if and only if
$A(e_{a1}, \cdots, e_{an}) = S\big(rep(e_{s1}), \cdots, rep(e_{sn})\big)$.

where $A(e_{a1}, \cdots, e_{an})$ is a formula of Ada service.

Definition:

An Ada implementation is said to satisfy its specification if and only if for any formula S of the specification its representing formula $rep(S)$ is provable in Ada service.

Most of the specification formulas have the form $S_i \rightsquigarrow S_o$. Based on the above definition, an Ada implementation satisfies the temporal formula $S_i \rightsquigarrow S_o$ if and only if $rep(S_i) \rightsquigarrow rep(S_o)$ is a theorem in Ada service.

5.3.3. Method for Proving Temporal Formulas of Ada Services

Temporal formulas of service specification include resource constraints, concurrency constraints, fairness policies, simultaneity policies, and user agreements. Each specification formula has a representing formula which must be proven in the Ada service. From the semantics of Ada tasking, the concurrency constraints are automatically satisfied assuming no variables are shared by tasks other than those shared for the rendezvous. Thus, the concurrency constraints of the service specifications need not to be verified.

(1) Verification of Resource Constraints:

The method for verification of resource constraints is straightforward. Suppose that the resource constraints of the service specification are formulas $R_{s1}, \cdots, R_{sn}$ then the proof steps are:

step 1. Develop a resource invariant formula R_a from the code of Ada service and from the semantics of Ada language.

step 2. Verify that the invariant formula R_a implies each of the representing formulas $rep(R_{s1}), \cdots, rep(R_{sn})$ of the Ada service.

(2) Verification of the Fairness Formulas

Generally, in Ada services, we use Ada tasks to synchronize user tasks during service access. Since the users waiting in an entry queue are accepted for service in FIFO order, we define:

$T_u \rightarrow Q_e[m]$ *if and only if user task* T_u *is the m-th user waiting in the queue* Q_e *of entry operation "e".*

Let T_s be the service task, and C represent the requested ***conditions*** for the entry operation "e" of the service task T_s. Then, a general *fairness* formula of Ada service can be expressed as:

$$\left((T_u \rightarrow T_s.Q_e[m]) \wedge \Box\Diamond C\right) \rightsquigarrow \left((T_s\ accept(e)) \wedge (T_u\ rendezvous(e))\right)$$

where $T_s.Q_e$ represents the entry queue Q_e of the service task T_s. $(T_s\ accept(e))$ means that the service task T_s has selected an entry operation "e" for the next execution and accepted the user task at the head of the entry queue Q_e for the next *rendezvous*. $(T_u\ rendezvous(e))$ means that the user task T_u begins the rendezvous with the service task. We also use the variable ***no_activity*** to denote that no entry operation is currently performing its computation for changing the resource state. Note that if currently no rendezvous takes place, then the variable ***no_activity*** is true. However, whenever the variable ***no_activity*** is true this does not imply that no rendezvous takes place currently. It is possible that an entry operation gets blocked during the rendezvous before changing the resource state.

Given the general *fairness* formula of the Ada service, if we let

$$I_o = \left((T_s\ accept(e)) \wedge (T_u\ rendezvous(e))\right),$$

then a *fairness* formula can be proven by induction as follows:

step 1. prove $(T_u \rightarrow Q_e[1]) \wedge \Box\Diamond C \rightsquigarrow I_o$;

step 2. suppose $(T_u \rightarrow Q_e[m]) \wedge \Box\Diamond C \rightsquigarrow I_o$,

then prove $(T_u \rightarrow Q_e[m+1]) \wedge \Box\Diamond C \rightsquigarrow I_o$.

(3) Verification of the Temporal Formula $I_i \rightsquigarrow I_o$

Most of temporal formulas of the Ada services representing formulas for expressing fairness policies, simultaneity policies, and user agreements have the form of $I_i \rightsquigarrow I_o$. Except for fairness formulas, other temporal formulas are proven by the following steps:

step 1. Assume the assertion $I_i \rightsquigarrow I_o$ is false; i.e., assume $\Diamond(I_i \wedge \Box\neg I_o)$.

step 2. Show a contradiction from the code of Ada service and from the semantics of Ada language.

5.4. IMPLEMENTATION VERIFICATION FOR THE RESOURCE ALLOCATOR

In this section we will verify formally the correctness of FIFO resource allocator whose implementation is given in Section 5.2. The verification method presented in Section 5.3 will be used to demonstrate that the FIFO_RESOURCE_ALLOCATOR package and its USER_TASK meet the specifications of resource allocator given in Section 4.5. For the formal verification of the general (non-FIFO) resource allocator, we can apply the same verification method as that used in this section. Thus, for the sake of brevity that verification is not included herein.

Based on the verification method, we begin by defining the representing entities for specification of the Ada service. Then we verify the validity of each representing formula in the implemented Ada service. The representation for each type, constant, variable, operator, and operation of specification of resource allocator in the Ada task FIFO_RESOURCE is given in Figure 6 below. Note that each operation has the user identity as a parameter, which although implicit in the specification, is explicitly expressed in the Ada service. For the sake of convenience, in the following proofs we still use the specification operators (i.e., $\geq, \leq, \wedge, \vee, \neg$, etc.) in the formulas of Ada service. However, we shall bear in mind that they are actually the Ada operators >=, <=, and, or, not, etc.

	Specification Entities e_s	Representing Entities $rep(e_s)$
Type	integer	INTEGER
	boolean	BOOLEAN
	userid	USERID
	units	UNITS
Constant	size	MAX
	quota	QUOTA
Variable	id	ID
	free	FREE
	own	OWN
operator	$>(s_1, s_2)$	$>(rep(s_1), rep(s_2))$
	$<(s_1, s_2)$	$<(rep(s_1), rep(s_2))$
	$\geq(s_1, s_2)$	$>=(rep(s_1), rep(s_2))$
	$\leq(s_1, s_2)$	$<=(rep(s_1), rep(s_2))$
	$\wedge(s_1, s_2)$	**and**$(rep(s_1), rep(s_2))$
	$\vee(s_1, s_2)$	**or**$(rep(s_1), rep(s_2))$
	$\neg(s)$	**not**$(rep(s))$
	$\Rightarrow(s_1, s_2)$	**if** $rep(s_1)$ **then** $rep(s_2)$
operation	Acquire(n)	ACQUIRE($rep(id), rep(n)$)
	Release(n)	RELEASE($rep(id), rep(n)$)

Fig. 6. Entity Representations of Resource Allocator

5.4.1. Verification of the Resource Constraints

step 1:

From the Ada task FIFO_RESOURCE and the semantics of Ada language, we develop the invariant formula $R_a = R_{a1} \wedge R_{a2} \wedge R_{a3} \wedge R_{a4}$, where

$$R_{a1} = \Box((\text{FREE} \geq 0) \wedge (\text{FREE} \leq \text{MAX})),$$

$$R_{a2} = \Box((\text{OWN(ID)} \geq 0) \wedge (\text{OWN(ID)} \leq \text{QUOTA(ID)})),$$

$$R_{a3} = \Big(((\text{FREE} = \text{N}) \wedge \Box(\neg(\textit{after}(\text{ACQUIRE})) \wedge \neg(\textit{after}(\text{RELEASE})))) \Rightarrow \Box(\text{FREE} = \text{N})\Big),$$

$$R_{a4} = \Big(((\text{OWN(ID)} = \text{M}) \wedge \Box(\neg(\textit{after}(\text{ACQUIRE(ID,)})) \wedge \neg(\textit{after}(\text{RELEASE(ID,)})))) \Rightarrow \Box(\text{OWN(ID)} = \text{M})\Big).$$

The fact that the formula R_a is an invariant assertion of Ada task FIFO_RESOURCE is based on the following reasons:

1) variables FREE and OWN are declared within the service task FIFO_RESOURCE and thus, their values cannot be changed out side the service task;
2) initially FREE = 0 and for all users OWN(ID) = 0 thus, R_a is true the first time FIFO_RESOURCE is executed;
3) whenever a user request for resources exceeds its quota or tries to release resources not acquired previously a QUOTA_ERROR exception is raised. Thus, if R_a is true before a rendezvous begin, then R_a will be true after the rendezvous is finished.

step 2:

From the specification of the resource allocator, let $R_{s1}, R_{s2}, R_{s3}, R_{s4}$ be the four resource constraints specified. We obtain their representing formulas as follows:

$$rep(R_{s1}) = \Box((\text{FREE} \geq 0) \wedge (\text{FREE} \leq \text{MAX})),$$

$$rep(R_{s2}) = \Box((\text{OWN(ID)} \geq 0) \wedge (\text{OWN(ID)} \leq \text{QUOTA(ID)})),$$

$$rep(R_{s3}) = ((\text{FREE} = \text{N}) \Rightarrow ((\text{FREE} = \text{N}) \text{ UNTIL } (\textit{after}(\text{ACQUIRE}) \vee \textit{after}(\text{RELEASE})))),$$

$$rep(R_{s4}) = ((\text{OWN(ID)} = \text{M}) \Rightarrow ((\text{OWN(ID)} = \text{M}) \text{ UNTIL } (\textit{after}(\text{ACQUIRE(ID,)}) \vee \textit{after}(\text{RELEASE(ID,)})))).$$

From the semantics of the UNTIL operator, it is clear that R_a implies $rep(R_{s1})$, $rep(R_{s2})$, $rep(R_{s3})$, and $rep(R_{s4})$. Thus, the Ada service satisfies the resource constraints of the specification of the resource allocator.

5.4.2. Verification of the Fairness Policies

The *fairness* policies of the resource allocator are:

$$\big(at(\text{Acquire}) \wedge \Box\Diamond\left((free \geq Acquire.n) \wedge (\#Active = 0)\right)\big) \rightsquigarrow after(\text{Acquire}); \qquad (F1)$$

$$\big(at(\text{Release}) \wedge \Box\Diamond(\#Active = 0)\big) \rightsquigarrow after(\text{Release}). \qquad (F2)$$

We must prove that the service task FIFO_RESOURCE satisfies both of the fairness policies. That is, we must prove the validity of the representing formulas for $(F1)$ and $(F2)$ in the service task. We only verify the fairness formula $(F1)$. The same method can be applied for the proof of fairness formula $(F2)$.

Fairness formula $(F1)$ has the form of $S_i \rightsquigarrow S_o$. By substituting entities of fairness formula $(F1)$ with their corresponding Ada entities, we obtain the representing formula $rep(S_i) \rightsquigarrow rep(S_o)$ with

$$rep(S_i) = \big((T_u\ at(T_s.\text{ACQUIRE(N)})) \wedge \Box\Diamond\left((\text{FREE} \geq \text{N}) \wedge no\_activity\right)\big),$$

$$rep(S_o) = \big(T_u\ after(T_s.\text{ACQUIRE(N)})\big).$$

where T_s represents the service task FIFO_RESOURCE and T_u represents the user task of T_s. In order to prove $rep(S_i) \rightsquigarrow rep(S_o)$, first consider the temporal formula $I_i \rightsquigarrow I_o$, where

$$I_i = \big((T_u \rightarrow Q_{ACQUIRE}[m]) \wedge \Box\Diamond\left((\text{FREE} \geq \text{N}) \wedge no\_activity\right)\big),$$

$$I_o = \big((T_s\ accept(\text{ACQUIRE})) \wedge (T_u\ rendezvous(\text{ACQUIRE})) \wedge \Diamond(\text{FREE} \geq \text{N})\big).$$

From the semantics of the Ada task, a call on a task entry operation will be put in its entry queue for future rendezvous. Thus, $rep(S_i) \rightsquigarrow I_i$ is obvious. Suppose I_o is true, then eventually FREE $\geq$ N when the user task T_u is accepted for rendezvous with service task T_s. Thus, from the Ada code of FIFO_RESOURCE, neither the *accept* operation (i.e., the execution statements between key word *accept* and *end*) nor the execution statements following the *accept* operation will be blocked. Thus, $I_o \rightsquigarrow rep(S_o)$ is also true.

We now apply the proof procedures for the fairness formula given in Section 5.3 to demonstrate $I_i(v) \rightsquigarrow I_o(v)$.

step 1:

Prove the following temporal formula:

$$\big((T_u \rightarrow Q_{ACQUIRE}[1]) \wedge \Box\Diamond C\big) \rightsquigarrow I_o \qquad (F3)$$

where $C = ((\text{FREE} \geq \text{N}) \wedge no\_activity)$. In order to prove (F3), we first prove the following temporal formula:

$$\big((T_u \rightarrow Q_{ACQUIRE}[1]) \wedge \Box\Diamond C\big) \rightsquigarrow \big(I_o \vee \big((T_u \rightarrow Q_{ACQUIRE}[1]) \wedge (\text{FREE} = \text{MAX})\big)\big) \qquad (F4)$$

If FREE = MAX, then (F4) is automatically true; otherwise, since $\Box\Diamond C$ is true, eventually there exists a positive number $k, k < \text{MAX}$, $k \geq \text{N}$, and FREE $= k$. The task selected for next rendezvous will be either T_u or other tasks in the RELEASE queue. Thus,

$$
\begin{aligned}
&((T_u \rightarrow Q_{ACQUIRE}[1]) \wedge \diamond((\text{FREE} = k) \wedge \textit{no\_activity})) \\
\rightsquigarrow\ &(I_o \vee ((T_u \rightarrow Q_{ACQUIRE}[1]) \wedge (T_s\ \textit{accept}(\text{RELEASE})) \wedge (\text{FREE} = k))) \\
\rightsquigarrow\ &(I_o \vee ((T_u \rightarrow Q_{ACQUIRE}[1]) \wedge (\text{FREE} = k + \text{RELEASE.N}))) \\
\rightsquigarrow\ &(I_o \vee ((T_u \rightarrow Q_{ACQUIRE}[1]) \wedge (\text{FREE} = \text{MAX})))
\end{aligned}
$$

Note that the above temporal formula expression is based on the following convenient notation:

$$
\begin{aligned}
&(P_1 \rightsquigarrow P_2 \rightsquigarrow \cdots \rightsquigarrow P_n) \\
\equiv &((P_1 \rightsquigarrow P_2) \wedge (P_2 \rightsquigarrow P_3) \wedge \cdots \wedge (P_{n-1} \rightsquigarrow P_n))
\end{aligned}
$$

Now, by applying the temporal theorem $((P_1 \rightsquigarrow P_2) \wedge (P_2 \rightsquigarrow P_3)) \Rightarrow (P_1 \rightsquigarrow P_3)$, we complete the proof of temporal formula (F4). Once FREE = MAX, the user task accepted for the next rendezvous must be the one at the head of the entry queue $Q_{ACQUIRE}$. Thus, the temporal formula $(F4)$ implies

$$
\left((T_u \rightarrow Q_{ACQUIRE}[1]) \wedge \Box\diamond C\right) \rightsquigarrow I_o.
$$

This completes the proof the proof of temporal formula $(F3)$.

step 2:

Assuming that $\left((T_u \rightarrow Q_{ACQUIRE}[m]) \wedge \Box\diamond C\right) \rightsquigarrow I_o$ is true, we want to prove

$$
\left((T_u \rightarrow Q_{ACQUIRE}[m+1]) \wedge \Box\diamond C\right) \rightsquigarrow I_o. \qquad (F5)
$$

Suppose $T'_u \rightarrow Q_{ACQUIRE}[m]$ when $T_u \rightarrow Q_{ACQUIRE}[m+1]$. Then

$$
\begin{aligned}
&((T_u \rightarrow Q_{ACQUIRE}[m+1]) \wedge \Box\diamond C) \\
\rightsquigarrow &\Big(((T_u \rightarrow Q_{ACQUIRE}[1]) \wedge \Box\diamond C)) \\
&\qquad\qquad \wedge (T_s\ \textit{accept}(\text{ACQUIRE})) \wedge (T'_u\ \textit{rendezvous}(\text{ACQUIRE}))\Big) \\
\rightsquigarrow &((T_u \rightarrow Q_{ACQUIRE}[1]) \wedge \Box\diamond C) \\
\rightsquigarrow &I_o \quad \text{which is implied by temporal formula } (F3).
\end{aligned}
$$

Thus, we have proven the temporal formula $(F5)$. Now, by induction, temporal formulas $(F3)$ and $(F5)$ imply the temporal formula $I_i \rightsquigarrow I_o$. We conclude that the representing formula $rep(S_i) \rightsquigarrow rep(S_o)$ is true.

5.4.3. Verification of the Simultaneity Policies

Let S_1 and S_2 be the first and second simultaneity formulas of the resource allocator respectively. By substituting each service specification entity with the corresponding Ada entity, we obtain $rep(S_1)$ and $rep(S_2)$, where

$$rep(S_1) = \big(rep(S_{i1} \rightsquigarrow rep(S_{o1})\big),$$

$$rep(S_2) = \big(rep(S_{i2} \rightsquigarrow rep(S_{o2})\big), \quad \text{and}$$

$$rep(S_{i1}) = \big((T_u \ in(T_s.\text{ACQUIRE(N)})) \wedge (\Box\Diamond(\text{FREE} \geq \text{N})) \wedge (\Box\Diamond\, no\_activity)\big),$$

$$rep(S_{o1}) = \big((\text{FREE} \geq \text{N}) \wedge no\_activity\big),$$

$$rep(S_{i2}) = \big((T_u \ in(T_s.\text{ACQUIRE(N)})) \wedge \Box\Diamond(T_s \ accept(\text{RELEASE}))\big),$$

$$rep(S_{o2}) = (\text{FREE} \geq \text{N}).$$

In the above formulas, T_s represents the service task FIFO_RESOURCE and T_u represents the user task of T_s.

A. Proof of $rep(S_1)$:

The representing formula $rep(S_1)$ can be proven true using the following line of reasoning:

1. The temporal formula $rep(S_{i1})$ implies

$$rep(S_{i1}) \Rightarrow \big(T_u \ in(T_s.\text{ACQUIRE(N)})\big),$$

$$rep(S_{i1}) \Rightarrow \Box\Diamond(\text{FREE} \geq \text{N}), \quad \text{and}$$

$$rep(S_{i1}) \Rightarrow \Box\Diamond(no\_activity).$$

2. From the code of task FIFO_RESOURCE, $\Box\Diamond(no\_activity)$ implies that service task T_s is waiting infinitely often for a rendezvous, or control in T_s is infinitely often at the beginning of the "while loop" within the ACQUIRE operation.
3. $\Box\Diamond(\text{FREE} \geq \text{N})$ implies that control in T_s cannot get blocked at the "while loop" when it begins a rendezvous with any user task that issues the call to ACQUIRE(N) for any value of N. Note that if a call to ACQUIRE(N) causes the QUOTA_ERROR exception because of an inadequate value of N, then control in T_s will not reach the "while loop".
4. User tasks waiting on an entry queue are selected for rendezvous in FIFO order. Thus, from the conclusion of point 3 above, once the user task T_u is waiting on the entry queue $Q_{ACQUIRE}$, it will eventually be selected for rendezvous. Otherwise, some other invocation to ACQUIRE must be blocked at the "while loop" which contradicts the conclusion of point 3.
5. When the service task T_s begins a rendezvous with T_u, either $rep(S_{o1})$ is true at the beginning of rendezvous or it will eventually be true when a large enough number of RELEASE calls have relinquished resources such that FREE $\geq$ N. We conclude that $rep(S_1)$ is true. (Note that we have excluded the

case when T_u causes exceptions during rendezvous. Otherwise, $rep(S_1)$ will not be true. However, this case does not cause denial of service because the service requested by T_u may not necessary be denied by other users.)

B. Proof of $rep(S_2)$:

Suppose $rep(S_2)$ is not true. Then

$$\begin{aligned}
&\neg rep(S_2) \\
&= \diamond \left(rep(S_{i2}) \wedge \square \neg rep(S_{o2}) \right) \\
&= \diamond \Big(\left(T_u \ in(T_s.\text{ACQUIRE(N)}) \right) \wedge \left(\square \diamond \left(T_s \ accept(\text{RELEASE}) \right) \right) \wedge \square(\text{FREE} < \text{N}) \Big).
\end{aligned}$$

Hence, the falseness of $rep(S_2)$ implies that

$$\diamond \Big(\left(T_u \ in(T_s.\text{ACQUIRE(N)}) \right) \wedge \left(\square(\text{FREE} < \text{N}) \right) \wedge \square \diamond \left(T_s \ accept(\text{RELEASE}) \right) \Big). \qquad (M1)$$

From the code of task FIFO_RESOURCE, a user cannot release resources that are not previously acquired. Otherwise, the service task will not accept the RELEASE call, and will raise an QUOTA_ERROR exception. Thus,

$$\left(\square \diamond \left(T_s \ accept(\text{RELEASE}) \right) \right) \Rightarrow \left(\square \diamond \left(T_s \ accept(\text{ACQUIRE}) \right) \right). \qquad (M2)$$

User tasks waiting on an entry queue are selected for rendezvous in FIFO order. Thus, temporal formulas $(M1)$ and $(M2)$ imply that the user task T_u will eventually be selected for rendezvous. From the code of task FIFO_RESOURCE, temporal formula $\square \diamond (T_s \ accept(\text{RELEASE}))$ also implies that T_u will not get blocked within the ACQUIRE operation. Thus, $\left(T_u \ in(T_s.\text{ACQUIRE(N)}) \right) \rightsquigarrow (\text{FREE} \geq \text{N})$. However, $(M1)$ implies $\diamond \big(\left(T_u \ in(T_s.\text{ACQUIRE(N)}) \right) \wedge \square(\text{FREE} < \text{N}) \big)$, which is a contradiction. We conclude that $rep(S_2)$ must be true.

5.4.4. Verification of the User Agreements

We shall prove that the Ada package FIFO_RESOURCE_ALLOCATOR (excluding the service task FIFO_RESOURCE) together with the user task USER satisfy the specification of the user agreement. We first obtain the representing formula by substituting the entities of the service specification with their corresponding representation entities. Let $S_i \rightsquigarrow S_o$ be the specification of user agreement and $rep(S_i) \rightsquigarrow rep(S_o)$ be the representing formula. Then

$$rep(S_i) = (T_u \ in(T_s.\text{ACQUIRE(N)})),$$
$$rep(S_o) = \Big((\Box\Diamond(T_s \ accept(\text{RELEASE}))) \vee (\text{FREE} \geq \text{N})\Big).$$

Suppose $rep(S_i) \rightsquigarrow rep(S_o)$ is not true. Then $\Diamond\big(rep(S_i) \wedge \Box\neg rep(S_o)\big)$, and we have

$$\Diamond\Big((T_u \ in(T_s.\text{ACQUIRE(N)})) \wedge (\Box(\text{FREE} < \text{N})) \wedge \Diamond\Box\neg(T_s \ accept(\text{RELEASE}))\Big). \qquad (U1)$$

The temporal formula $(U1)$ implies $\Diamond\Box\neg(T_s \ accept(\text{RELEASE}))$. This formula is true if and only if

1. no other user tasks issue RELEASE calls some time after the user task T_u has called ACQUIRE, or
2. all calls to RELEASE are either not accepted by the service (i.e., they are waiting on the entry queue $Q_{RELEASE}$ forever) or rejected by the QUOTA_ERROR exception.

Case 1 is not true for the following reasons. First, the temporal formula $(U1)$ implies $\Diamond\Box(\text{FREE} < \text{N})$. Thus, some users must have called ACQUIRE successfully and have not called RELEASE before T_u calls ACQUIRE. Second, from the code of user task USER, a RELEASE call always follows a ACQUIRE call. Thus, eventually some RELEASE calls will be issued after T_u called ACQUIRE.

Case 2 is not true for the following reasons. First, the service task will not always accept user tasks on $Q_{ACQUIRE}$ for rendezvous; e.g., when the resource pool is empty. Second, by using the access type tasks (i.e., the ACQUIRE_AGENT and the REL_AGENT) to represent users outside the package FIFO_RESOURCE_ALLOCATOR, the waiting RELEASE calls will not be aborted even when the client task has been aborted. Third, a RELEASE call will not cause QUOTA_ERROR exception if the client task has allocated the resources because a user task always release the same amount of resources as that acquired before.

Since both case 1 and 2 are not true, the temporal formula $\Diamond\Box\neg(T_s \ accept(\text{RELEASE}))$ cannot be true, which is a contradiction. Therefore, we have completed the proof.

6. CONCLUSIONS

A formal specification and verification method for the prevention of denial of service in shared services has been presented. The method has been applied to services written in Ada, using the "package" and "tasking" concepts. The method includes a service model, the notion of user agreements, and a temporal logic language. We also showed how the formal service specifications can be mapped in Ada service and how the formal proofs of specification can be carried out on Ada code. We conclude that the methods presented herein are practical and can be used for Ada services as well as for services written in other languages. Significantly more experience is necessary with the application of this method to distributed services (e.g., network access protocols, etc.) before the method can gain wide-spread use.

7. References

Berry, D.M. 1987. Towards a formal basis for the formal development method and the Ina Jo specification language. *IEEE Transactions on Software Engineering* SE-13/2 (February): 184-201.

Dijkstra, E.W. 1972. A class of allocation strategies inducing bounded delays only. In *SJCC,* 993-936. Montvale, NJ: AFIPS Press.

Department of Defense. *Trusted computer system evaluation criteria.* Washington, D.C.: U.S. DoD. CSC-STD-001-83.

Gligor, V.D. 1983. A note on the denial-of-service problem. In *Proceedings of the IEEE symposium on computer security and privacy, Oakland, CA, April 1983,* 5101-5111.

Gligor, V.D. 1983a. The verification of the protection mechanisms of high-level language machines. *International Journal of Computer and Information Sciences* 12/4 (August): 211-246.

Gligor, V.D. 1984. A note on denial-of-service in operating systems. *IEEE Transactions on Software Engineering* SE-10/3 (May): 320-324.

Goos, G. and J. Hartmanis, eds. 1983. *The programming language Ada reference manual.* Lecture Notes in Computer Science Series, vol. 155. New York: Springer-Verlag.

Gouda, M. 1979. Analysis of real-time control systems by the model of packet nets. In *Proceedings of the 1979 national computer conference, New York, 4-7 June 1979.* Montvale, NJ: AFIPS Press.

Gray, J.N. 1978. Notes on data base operating systems. In *Operating systems: An advanced course,* 393-481. Lecture Notes in Computer Science Series, vol. 60. New York: Springer-Verlag.

Habermann, A.N. 1969. Prevention of system deadlocks. *Communications of the ACM* 12/7 (July): 373-377.

Habermann, A.N., L. Flon, and L. Cooprider. 1976. Modularization and hierarchy in a family of operating systems. *Communications of the ACM* 19/5 (May): 266-272.

Hailpern, B.T. 1982. *Verifying concurrent process using temporal logic.* Lecture Notes in Computer Science Series, vol. 129. New York: Springer-Verlag.

Hailpern, B.T. and S.S. Owicki. 1983. Modular verification of computer communication protocols. *IEEE Transactions on Communications* COM-31/1 (January): 56-68.

Havender, J.W. 1968. Avoiding deadlock in multitasking systems. *IBM System Journal* 7/2: 74-84.

Hoare, C.A.R. 1972. Proof of correctness of data representations. *Acta Informatica* 1/4: 271-281.

Hoare, C.A.R. 1974. Monitors: An operating system structuring concept. *Communications of the ACM* 17/10 (October): 549-557.

Janson, P. 1976. *Using type extension to organize virtual memory mechanisms.* Cambridge, MA: Massachusetts Institute of Technology. MIT/LCS/TR-167.

Karp, R.A. 1984. Proving failure-free properties of concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems* 6/2 (April): 239-253.

Lamport, L. 1980. 'Sometime' is sometimes 'not never': On the temporal logic of programs. In *Proceedings of the 7th ACM symposium on the principles of programming languages, January 1980,* 174-185.

Lamport, L. 1983. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems* 5/2 (April): 190-222.

Lamport, L. 1986. *A simple approach to specifying concurrent systems.* Palo Alto, CA: Digital Equipment Corporation. System Research Center Report No. 15.

Lampson, B.W. and H. Sturgis. 1981. Atomic transactions. In *Distributed systems: Architecture and implementation,* edited by Lampson and Siegert. Lecture Notes in Computer Science Series, vol. 105. New York: Springer-Verlag.

Landwehr, C.E., and C.L. Heltmayer, and J. McLean. 1984. A security model for military message systems. *ACM Transactions on Computer Systems* 2/3 (August): 198-222.

Levin, R., E.S. Cohen, W.M. Corwin, F.J. Pollack, and W.A. Wulf. 1975. In *Proceedings of the 5th ACM symposium on operating systems principles,* 132-140.

Liskov, B., M. Herlihy, and L. Gilbert. 1986. Limitations fo synchronous communciation with static process structure in language for distributed computing. In *SIGACT & SIGPLAN 13th annual ACM symposium on principles of progrmaming languages, St. Petersburg Beach, FL, January 1986,* 150-159.

Millen, J.K. and C.M. Cerniglia. 1984. *Computer security models.* Bedford, MA: MITRE Corporation. Working Paper WP-25068.

Neumann, P.G., L. Robinson, K. Levitt, R.S. Boyer, and A. Saxena. 1975. *A provably secure operating system.* Palo Alto, CA: Stanford Research Institute. Final report, Stanford Research Institute Project 2581.

Osterhout, J.K., D.A. Scelza, and P. Sindhu. 1980. Medusa: An experience in distributed operating system structure. *Communications of the ACM* 23/2 (February): 92-105.

Owicki, S.S. 1979. Specification and proofs for abstract data types in concurrent programs. In *Program Construction,* 174-197. Lecture Notes in Computer Science Series, vol. 69. New York: Springer-Verlag.

Owicki, S.S. and L. Lamport. 1982. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems* 4/3 (July): 455-495.

Parnas, D.L. 1976. *Some hypothesis about the 'uses' hierarchy for operating systems.* Darmstadt, GFR: Technical University, Department of Informatics. Technical Report BS I 76/1.

Parnas, D.L. 1979. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering* SE-5/2 (March): 128-138.

Pnueli, A. 1977. The temporal logic of programs. In *Proceedings of the IEEE 18th annual symposium on the foundations of computer science, November 1977,* 46-57.

Pnueli, A. 1979. The temporal semantics of concurrent programs. In *Semantics of concurrent computation,* 1-20. Lecture Notes in Computer Science Series, vol. 70. New York: Springer-Verlag.

Pnueli, A. 1980. On the temporal analysis of fairness. In *Proceedings of the 7th annual symposium on the principles of programming languages, January 1980,* 163-173.

Popek, G.J. and C.S. Kline. 1975. A verifiable protection sytem. In *Proceedings of the first international conference on reliable software, Los Angeles, CA, March 1975,* 294-304.

Ramamritham, K. and R.M. Keller. 1983. Specification of synchronizing processes. *IEEE Transactions on Software Engineering* SE-9/6 (November): 722-733.

Ramamritham, K. 1985. Synthesizing code for resource controllers. *IEEE Transactions on Software Engineering* SE-11/8 (August): 774-783.

Ritchie, D.M. 1978. On the security of UNIX. *UNIX programmer's manual.* Vol. 2.

Scheid, J., S. Anderson, R. Martin, and S. Holtsberg. 1986. *The Ina Jo specification language reference manual.* Santa Monica, CA: System Development Corporation. Technical Report TM-(L)-6021/001/02.

Schroeder, M., D.D. Clark, and J.H. Saltzer. 1981. The Multics kernel design project. In *Proceedings of the 6th symposium on operating systems principles, West Lafayette, IN, November 1977.*

Schwartz, R.L. and P.M. Melliar-Smith. 1981. Temporal logic specification of distributed systems. In *Proceedings of the IEEE conference on distributed systems, April 1981,* 446-454.

Schwartz, R.W. and P.M. Melliar-Smith. 1982. From state machine to temporal logic: Specification methods for protocol standards. *IEEE Transactions on Communications* COM-30/12 (December): 2486-2496.

Walker, B.J., R.A. Kemmerer, and G.J. Popek. 1980. Specification and verification of the UCLA UNIX security kernel. *Communications of the ACM* 23/2 (February): 118-131.

Wegner, P. and S.A. Smolka. 1983. Processes, tasks, and monitors: A comparative study of concurrent programming primitives. *IEEE Transactions on Software Engineering* SE-9/4 (July): 446-462.

Welsh, J. and A. Lister. 1981. A comparative study of task communication in Ada. *Software–Practice and Experience* 11/3 (March): 257-290.

Wilkinson, et al. 1981. A penetration analysis of Burroughs large system. *Operating Systems Review* 15/1.

Wulfe, W.A., R.L. London, and M. Shaw. 1976. An introduction to the construction and verification of Alphard programs. *IEEE Transactions on Software Engineering* SE-2/4 (December): 253-265.

Yu, C.F. and V.D. Gligor. 1985. *Some examples of denial of service in communication networks.* College Park, MD: University of Maryland, Department of Electrical Engineering. Technical Report.

Appendix 1

Semantics of Temporal Logic

The use of *temporal logic* in program specification and verification was first proposed by Pnueli [Pnueli77], [Pnueli79]. Since then, several versions of *temporal logic* were proposed in the literature [Lamport80], [Schwartz81], [Owicki82], [Hailpern82], [Schwartz82], [Lamport83], [Ramamritham83]. *Temporal logic* adds operators to the standard logic system for reasoning about future *progress* of program computations. A computation is the sequence of states that results from program execution. An operation in the service has a definite state at any instant of time. The behavior of the user within a service is completely described by the evolution of the state with time. If concurrency is allowed within a service, the execution state sequence is modeled by a nondeterministic interleaving of execution of individual user processes within that service. In general, a *temporal logic* formula is considered with respect to a reference state called the "now". A *temporal logic* formula expresses how program properties can change between the reference state and the set of states, called the "future", that are accessible from the reference state.

Since *temporal logic* is an extension of predicate calculus, the usual logic operator $\wedge$ (and), $\vee$ (or), $\neg$ (negation), and $\Rightarrow$ (implication) can be included in a *temporal logic* formula. In general, a *temporal logic* formula is constructed from a set of predicates, the usual logic operator, and the *temporal* operators $\Box$, $\Diamond$, and UNTIL. A predicate is a boolean function of a computation state. The unary operation $\Box$ is pronounced "henceforth". Let P be a predicate, the formula $\Box$P means "P is true now and will remain true for all future states in the computation". The unary operator $\Diamond$ is pronounced "*eventually*". The formula $\Diamond$P means "P" is true now or will become true sometime in the future". The operators $\Box$ and $\Diamond$ are dual, that is,

$$\Box P \equiv \neg \Diamond \neg P \quad \text{or} \quad \Diamond P \equiv \neg \Box \neg P$$

with these unary *temporal* operator, many properties can be stated. For example, the progress property of an operation "op" can be expressed by the formula:

$$at(\text{op}) \Rightarrow \Diamond after(\text{op})$$

Where *at*(op) and *after*(op) are predicates used to keep track of control position of an invocation. The predicate *at*(op) is true if and only if control is at the entry point to "op", and *after*(op) is true if and only if control is at the exit point of "op". The predicate *in*(op) must also be introduced. The predicate *in*(op) is true if and only if control is anywhere inside operation "op", including its entry point but excluding the exit point. Hence, if control is currently at the entry point to "op" and never reaches the exit point thereafter, then control will remain in "op" forever, i.e.,

$$\big(at(\text{op}) \wedge \Box \neg after(\text{op})\big) \Rightarrow \Box in(\text{op}).$$

Another useful *temporal* formula, which states that a property P always causes another property Q to become true subsequently, can be expressed by:

$$P \rightsquigarrow Q \equiv \Box(P \Rightarrow \Diamond Q)$$

The combination of these two unary *temporal* operators is also useful. For example, to express that a property P is satisfied "*infinitely often*", we can use the formula $\Box \Diamond$ P (*infinitely often* P). The $\Box\Diamond$ operator is especially useful for expressing the *fairness* property within a service whenever concurrent access is allowed. In particular, if condition P ever becomes false, P is guaranteed to become true again at some later time. Consequently, if several user operations are simultaneously waiting for the same condition P within a service, formula $\Box \Diamond$ P states that these operations will *eventually* pass the condition. The dual of $\Box\Diamond$ operator is $\Diamond\Box$ (i.e., $\Diamond\Box \equiv \neg\Box \Diamond \neg$). The formula $\Diamond\Box$P (*eventually always* P) states that there is some point in the future at which P becomes true and remains true thereafter. The $\Diamond\Box$ operator is useful for expressing *lack-of-progress* properties. For example, *deadlock* between operation op1 and operation op2 within a service can be expressed by the formula:

$$\Diamond\Box(in(\text{op1}) \wedge in(\text{op2}))$$

The binary *temporal* operator UNTIL is used to express relationships between two points in a computation (i.e., to express ordering property). The formula P UNTIL Q means "P is true for all states until the first state where Q is true"; i.e.,

P UNTIL Q ≡ P remains true until Q becomes true.

This formula does not express the value of P when Q becomes true. □P can be derived in terms of the UNTIL operator as:

□P ≡ P UNTIL false.

UNTIL operator is generally used for expressing formulas of the form P ⇒ (P UNTIL Q), stating that if P is true "now", it will remain true until Q becomes true. Note that the definition of the UNTIL operator in this paper does not assert the eventuality property; i.e., P UNTIL Q does not imply ◇Q, which is different from that of reference [Pnueli 80].

Appendix 2

Derived Temporal Theorems

This appendix begins by presenting the syntax of temporal formulas and then derives a list of temporal theorems that are used in this paper. Some portions of this appendix have been taken from Hailpern's report [Hailpern82].

1. The Syntax of Temporal Formulas

A description of a temporal logic system includes four parts: a list of atomic predicates: a set of formation rules that define which predicates defined in terms of the atomic predicates are *(well-formed) formulas*; a set of formulas, known as axioms; and a set of inference rules that permit operations on the axioms and those formulas that have been derived from previous applications of the inference rules. The formulas obtained by applications of inference rules are known as *theorems*. If formula P is an axiom or a theorem then we write $\vdash P$. The syntax of temporal formulas is based on the following rules and axioms:

1) Formation Rules:

- an atomic predicate is a formula;
- if A is a formula then so are $\neg A, \Box A$, and $\Diamond A$;
- if A and B are formulas then so are $(A \vee B), (A \wedge B), (A \Rightarrow B)$, and $(A \equiv B)$.

2) Temporal Axioms:

Let P and Q are formulas, we have the following temporal axioms:

$(A1): \quad \Diamond P \equiv \neg\Box\neg P,$ $\qquad (A1'): \quad \Box P \equiv \neg\Diamond\neg P;$

$(A2): \quad \Box P \Rightarrow P,$ $\qquad (A2'): \quad P \Rightarrow \Diamond P;$

$(A3): \quad \Box(P \Rightarrow Q) \Rightarrow (\Box P \Rightarrow \Box Q),$ $\qquad (A3'): \quad (\Box P \wedge \Diamond Q) \Rightarrow \Diamond(P \wedge Q);$

$(A4): \quad (P \text{ UNTIL } Q) \Rightarrow (\Box\neg Q \Rightarrow \Box P);$

$(A5): \quad (P \rightsquigarrow Q) \equiv \Box(P \Rightarrow \Diamond Q).$

3) Inference Rules:

$$(I1): \quad \frac{(\vdash P) \wedge (\vdash (P \Rightarrow Q))}{\vdash Q};$$

$$(I2): \quad \frac{\vdash P}{\vdash \Box P};$$

$$(I3): \quad \frac{\vdash (P \equiv Q)}{\vdash (f(P) \equiv f(Q))}.$$

2. The Derived Theorems

1) Basic Derived Theorems:

The following theorem has been proven in [Hailpern82]:

$(T1): \quad \Box\Box P \equiv \Box P,$ $\qquad (T1'): \quad \Diamond\Diamond P \equiv \Diamond P;$

$(T2): \quad \Box\Diamond\Box P \equiv \Diamond\Box P,$ $\qquad (T2'): \quad \Diamond\Box\Diamond P \equiv \Box\Diamond P;$

$(T3): \quad \Box(P \wedge Q) \equiv (\Box P \wedge \Box Q),$ $\qquad (T3'): \quad \Diamond(P \vee Q) \equiv (\Diamond P \vee \Diamond Q);$

$(T4): \quad \Diamond\Box(P \wedge Q) \equiv (\Diamond\Box P \wedge \Diamond\Box Q),$ $\qquad (T4'): \quad \Box\Diamond(P \vee Q) \equiv (\Box\Diamond P \vee \Box\Diamond Q);$

$(T5): \quad \Box(P \vee Q) \Rightarrow (\Box\Diamond P \vee \Box\Diamond Q),$ $\qquad (T5'): \quad (\Diamond\Box P \wedge \Diamond\Box Q) \Rightarrow \Diamond(P \wedge Q);$

$(T6): \quad \Box(P \vee Q) \Rightarrow (\Box P \vee \Diamond Q),$ $\qquad (T6'): \quad (\Diamond P \wedge \Box Q) \Rightarrow \Diamond(P \wedge Q);$

$(T7): \quad (P \Rightarrow Q) \Rightarrow ((P \vee R) \Rightarrow (Q \vee R))$

2) Other Derived Theorems:

The following theorems are used in this paper. Each theorem has a proof followed.

$(D1): \quad \Box(\neg P \Rightarrow \Diamond P) \Rightarrow \Box\Diamond P$

$$\Box(\neg P \Rightarrow \Diamond P) \equiv \Box(P \vee \Diamond P) \qquad (I3)$$
$$\Box(P \vee \Diamond P) \Rightarrow (\Box\Diamond P \vee \Box\Diamond\Diamond P) \qquad (T5)$$
$$\Box(P \vee \Diamond P) \Rightarrow (\Box\Diamond P \vee \Box\Diamond P) \qquad (T1', I3)$$
$$\Box(P \vee \Diamond P) \Rightarrow \Box\Diamond P \qquad (I3)$$
$$\Box(\neg P \Rightarrow \Diamond P) \Rightarrow \Box\Diamond P \qquad (I3)$$

$(D2): \quad ((P \Rightarrow Q) \wedge ((P \wedge Q) \Rightarrow R)) \Rightarrow (P \Rightarrow R)$

$$((P \Rightarrow Q) \wedge ((P \wedge Q) \Rightarrow R)) \equiv ((\neg P \vee Q) \wedge (\neg P \vee \neg Q \vee R)) \qquad (I3)$$
$$((P \Rightarrow Q) \wedge ((P \wedge Q) \Rightarrow R)) \equiv (\neg P \vee (Q \wedge (\neg Q \vee R))) \qquad (I3)$$
$$((P \Rightarrow Q) \wedge ((P \wedge Q) \Rightarrow R)) \equiv (\neg P \vee (Q \wedge \neg Q) \vee (Q \wedge R)) \qquad (I3)$$
$$((P \Rightarrow Q) \wedge ((P \wedge Q) \Rightarrow R)) \equiv (\neg P \vee (Q \wedge R)) \qquad (I3)$$
$$((P \Rightarrow Q) \wedge ((P \wedge Q) \Rightarrow R)) \equiv ((\neg P \vee Q) \wedge (\neg P \vee R)) \qquad (I3)$$
$$((\neg P \vee Q) \wedge (\neg P \vee R)) \Rightarrow (\neg P \vee R)$$
$$((P \Rightarrow Q) \wedge ((P \wedge Q) \Rightarrow R)) \Rightarrow (P \Rightarrow R) \qquad (I3)$$

$(D3): \quad (\Box(P \Rightarrow Q) \wedge \Box((P \wedge Q) \Rightarrow R)) \Rightarrow \Box(P \Rightarrow R)$

$$\Box(((P \Rightarrow Q) \wedge ((P \wedge Q) \Rightarrow R)) \Rightarrow (P \Rightarrow R)) \qquad (D2, I2)$$
$$\Box((P \Rightarrow Q) \wedge ((P \wedge Q) \Rightarrow R)) \Rightarrow \Box(P \Rightarrow R) \qquad (A3)$$
$$(\Box(P \Rightarrow Q) \wedge \Box((P \wedge Q) \Rightarrow R)) \Rightarrow \Box(P \Rightarrow R) \qquad (T3, I3)$$

$(D4): \quad (P \rightsquigarrow Q) \Rightarrow (\Diamond\Box P \Rightarrow \Box\Diamond Q)$

$$(P \rightsquigarrow Q) \equiv \Box(P \Rightarrow \Diamond Q) \qquad (A5)$$
$$(P \rightsquigarrow Q) \equiv \Box(\neg P \vee \Diamond Q) \qquad (I3)$$
$$\Box(\neg P \vee \Diamond Q) \Rightarrow ((\Box\Diamond \neg P) \vee \Box\Diamond\Diamond Q) \qquad (T5)$$
$$(P \rightsquigarrow Q) \Rightarrow ((\Box\Diamond \neg P) \vee \Box\Diamond\Diamond Q) \qquad (T1', I3)$$
$$(P \rightsquigarrow Q) \Rightarrow ((\neg\Diamond \Box P) \vee \Box\Diamond\Diamond Q) \qquad (A1, A1', I3)$$
$$(P \rightsquigarrow Q) \Rightarrow (\Diamond\Box P \Rightarrow \Box\Diamond Q) \qquad (I3)$$

$(D5): \quad \Box(P \Rightarrow \Diamond Q) \equiv \Box(\Diamond P \Rightarrow \Diamond Q)$

$\Rightarrow$

$$\Box(P \Rightarrow \Diamond Q) \equiv \Box(\neg P \vee \Diamond Q) \qquad (I3)$$
$$\Box(\neg P \vee \Diamond Q) \Rightarrow (\Box\neg P \vee \Diamond Q) \qquad (T6)$$
$$\Box(P \Rightarrow \Diamond Q) \Rightarrow (\neg\Diamond P \vee \Diamond Q) \qquad (I3, A1')$$
$$\Box(P \Rightarrow \Diamond Q) \Rightarrow (\Diamond P \Rightarrow \Diamond Q) \qquad (I3)$$
$$\Box(\Box(P \Rightarrow \Diamond Q) \Rightarrow (\Diamond P \Rightarrow \Diamond Q)) \qquad (I2)$$
$$(\Box\Box(P \Rightarrow \Diamond Q)) \Rightarrow \Box(\Diamond P \Rightarrow \Diamond Q) \qquad (A3)$$
$$\Box(P \Rightarrow \Diamond Q) \Rightarrow \Box(\Diamond P \Rightarrow \Diamond Q) \qquad (A3)$$

$\Leftarrow$

$$
\begin{aligned}
&(\diamond P \Rightarrow \diamond Q) \equiv (\neg \diamond P \vee \diamond Q) \\
&(\diamond P \Rightarrow \diamond Q) \equiv (\boxdot \neg P \vee \diamond Q) && (A1') \\
&\boxdot \neg P \Rightarrow \neg P && (A2) \\
&(\boxdot \neg P \vee \diamond Q) \Rightarrow (\neg P \vee \diamond Q) && (T7) \\
&(\diamond P \Rightarrow \diamond Q) \Rightarrow (P \Rightarrow \diamond Q) && (I3) \\
&\boxdot((\diamond P \Rightarrow \diamond Q) \Rightarrow (P \Rightarrow \diamond Q)) && (I2) \\
&\boxdot(\diamond P \Rightarrow \diamond Q) \Rightarrow \boxdot(P \Rightarrow \diamond Q) && (A3)
\end{aligned}
$$

$(D6): \quad ((P \rightsquigarrow Q) \wedge (Q \rightsquigarrow R)) \Rightarrow (P \rightsquigarrow R)$

$$
\begin{aligned}
&((P \Rightarrow \diamond Q) \wedge (\diamond Q \Rightarrow \diamond R)) \Rightarrow (P \Rightarrow \diamond R) \\
&\boxdot((P \Rightarrow \diamond Q) \wedge (\diamond Q \Rightarrow \diamond R)) \Rightarrow (P \Rightarrow \diamond R)) && (I2) \\
&\boxdot((P \Rightarrow \diamond Q) \wedge (\diamond Q \Rightarrow \diamond R)) \Rightarrow \boxdot(P \Rightarrow \diamond R)) && (A3) \\
&(\boxdot(P \Rightarrow \diamond Q) \wedge \boxdot(\diamond Q \Rightarrow \diamond R)) \Rightarrow \boxdot(P \Rightarrow \diamond R)) && (T3, I3) \\
&(\boxdot(P \Rightarrow \diamond Q) \wedge \boxdot(Q \Rightarrow \diamond R)) \Rightarrow \boxdot(P \Rightarrow \diamond R)) && (D5, I3) \\
&((P \rightsquigarrow Q) \wedge (Q \rightsquigarrow R)) \Rightarrow (P \rightsquigarrow R) && (A5, I3)
\end{aligned}
$$

Distribution List for IDA Paper P-2120

NAME AND ADDRESS	NUMBER OF COPIES
Sponsor	
Dr. John Solomond Director Ada Joint Program Office Room 3E114, The Pentagon Washington, D.C. 2030-3081	2
Mr. John Faust Rome Air Development Center RADC/COTC Griffiss AFB, NY 13441	2
Other	
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dr. Virgil D. Gligor EE Dept College of Engineering University of Maryland College Park, MD 20742	2
Dr. Che-Fn Yu GTE Laboratories, Inc. 40 Sylvan Road Waltham, MA 02254	2
IIT Research Institute 4600 Forbes Blvd., Suite 300 Lanham, MD 20706 Attn. Ann Eustice	1
Mr. Karl H. Shingler Department of the Air Force Software Engineering Institute Joint Program Office (ESD) Carnegie Mellon University Pittsburgh, PA 15213-3890	1

NAME AND ADDRESS	NUMBER OF COPIES
CSED Review Panel	
Dr. Dan Alpert, Director Program in Science, Technology & Society University of Illinois Room 201 912-1/2 West Illinois Street Urbana, Illinois 61801	1
Dr. Thomas C. Brandt 10302 Bluet Terrace Upper Marlboro, MD 20772	1
Dr. Ruth Davis The Pymatuning Group, Inc. 2000 N. 15th Street, Suite 707 Arlington, VA 22201	1
Dr. C.E. Hutchinson, Dean Thayer School of Engineering Dartmouth College Hanover, NH 03755	1
Mr. A.J. Jordano Manager, Systems & Software Engineering Headquarters Federal Systems Division 6600 Rockledge Dr. Bethesda, MD 20817	1
Dr. Ernest W. Kent Philips Laboratories 345 Scarborogh Road Briarcliff Manor, NY 10510	1
Dr. John M. Palms, President Georgia State University University Plaza Atlanta, GA 30303	1
Mr. Keith Uncapher University of Southern California Olin Hall 330A University Park Los Angeles, CA 90089-1454	1

NAME AND ADDRESS	NUMBER OF COPIES
IDA	
General W.Y. Smith, HQ	1
Mr. Philip L. Major, HQ	1
Dr. Robert E. Roberts, HQ	1
Mr. John Boone, CSED	1
Ms. Anne Douville, CSED	1
Mr. Terry Mayfield, CSED	1
Ms. Katydean Price, CSED	2
Dr. J. Eric Roskos, CSED	1
Mr. Steve R. Welke, CSED	1
Dr. Richard L. Wexelblat, CSED	1
IDA Control & Distribution Vault	3